C#.NET SIMPLIFIED

Mohammed Manzoor Ahmed
www.manzoorthetrainer.com

CONTENTS

ABOUT THE AUTHOR **4**

CHAPTER 1: ARITHMETIC OPERATIONS **5**

CHAPTER 2: TYPE CASTING **15**

CHAPTER 3: CONTROL STRUCTURES **23**

1. CONDITIONAL: **24**

IF AND IF-ELSE CONDITION: 24

ELSE-IF LADDER: 29

SWITCH CASE: 36

2. ITERATIVE STATEMENTS OR LOOPS: **41**

FOR LOOP: 41

WHILE LOOP: 48

DO-WHILE LOOP: 53

CHAPTER 4: ARRAYS **57**

CHAPTE 5 : UNDERSTANDING THE CONCEPT OF FOREACH LOOP AND SYSTEM.ARRAY METHODS IN C#. **64**

CHAPTER 6 : UNDERSTANDING THE CONCEPT OF STRUCTURES **75**

CHAPTER 7 : CLASS **90**

CHAPTER 8 : METHOD OVERLOADING OR STATIC POLYMORPHISM **99**

CHAPTER 9 : THIS KEYWORD IN C#: **106**

CHAPTER 10 : STATIC VARIABLE AND STATIC CONSTRUCTOR. **116**

CHAPTER 11 : STATIC METHODS AND STATIC METHOD OVERLOADING **122**

CHAPTER 12 : PROPERTIES IS C# **129**

CHAPTER 13 : NAMESPACES IN C# **140**

CHAPTER 14 : UNDERSTANDING THE CONCEPT OF INHERITANCE AND PROTECTED VARIABLES IN C#. 148

CHAPTER 15 : CONSTRUCTOR CHAINING IN C#. 155

CHAPTER 16 : METHOD OVERRIDING, VARIOUS TYPES OF INHERITANCE AND CONSTRUCTOR CHAINING IN C#. 168

CHAPTER 17 : ABSTRACT METHOD AND ABSTRACT CLASS IN C#. 184

CHAPTER 18 : REAL-TIME RUNTIME POLYMORPHISM IMPLEMENTATION IN C#. 191

CHAPTER 19 : SEALED METHOD AND SEALED CLASS IN C#. 207

CHAPTER 20 : INTERFACES AND MULTIPLE INHERITANCE IN C#. 213

CHAPTER 21 : COLLECTION CLASSES AND ITS LIMITATION IN C#. 222

CHAPTER 22 : GENERIC COLLECTION CLASSES IN C#. 232

CHAPTER 23 : CONCEPT OF DELEGATES, UNICAST DELEGATES AND MULTICAST DELEGATES IN C#. 239

CHAPTER 24 : DELEGATES, CALLBACK AND METHOD CONCEPT IN C#. 247

ABOUT THE AUTHOR

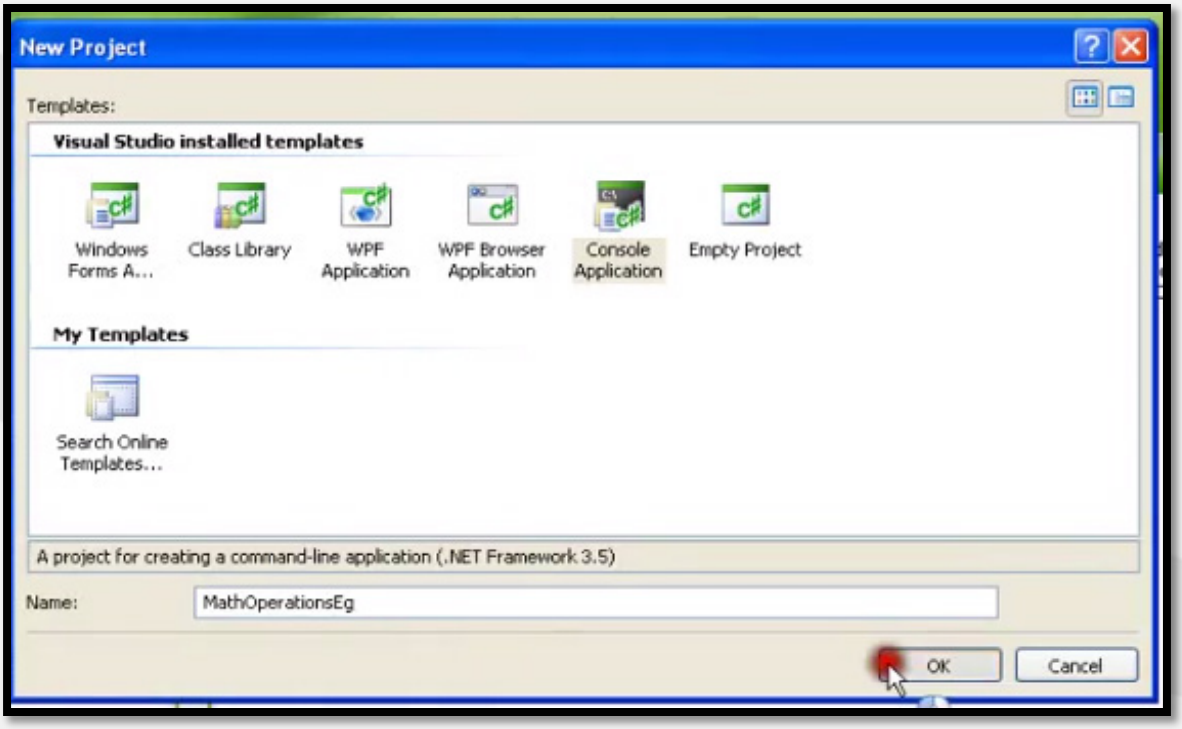
Manzoor is a Microsoft Certified Trainer who has been working on MS .Net technologies for more than a decade. Apart from development he is also passionate about delivering training on various MS .Net technologies and he has 10+ years of experience as a software development teacher. He writes articles for code-project as well. His YouTube channel has 1 million hits. He is the founder of ManzoorTheTrainer portal.

"I focus on simplifying, complex concepts..."

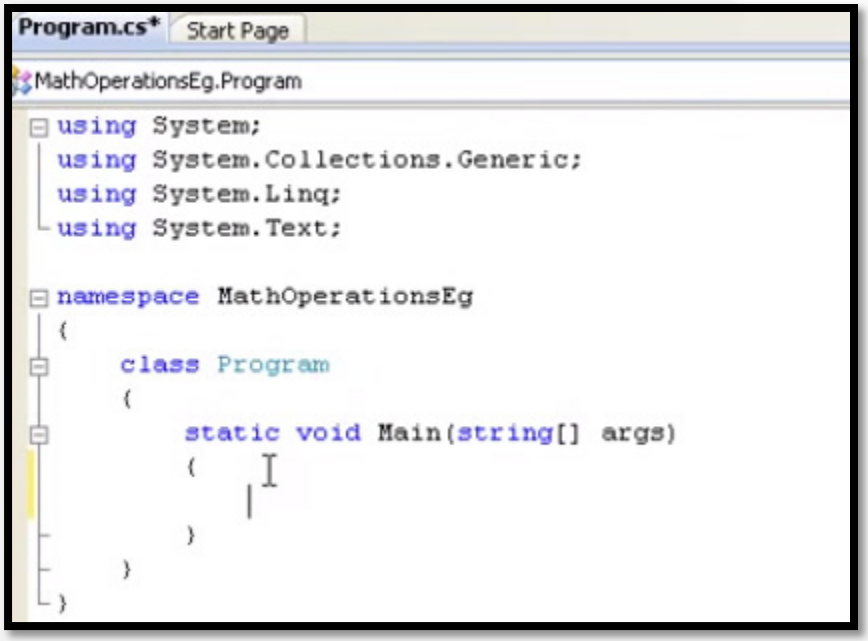
– ManzoorTheTrainer

CHAPTER 1: ARITHMETIC OPERATIONS

- In this chapter we are going to see few arithmetic operations that we can perform using C#.
- Arithmetic operatios like adding, subtracting, division, multiplication etc.
- Goto programs→Visual Studio.
- Select new project→Console application and name the project.
- Select Ok.



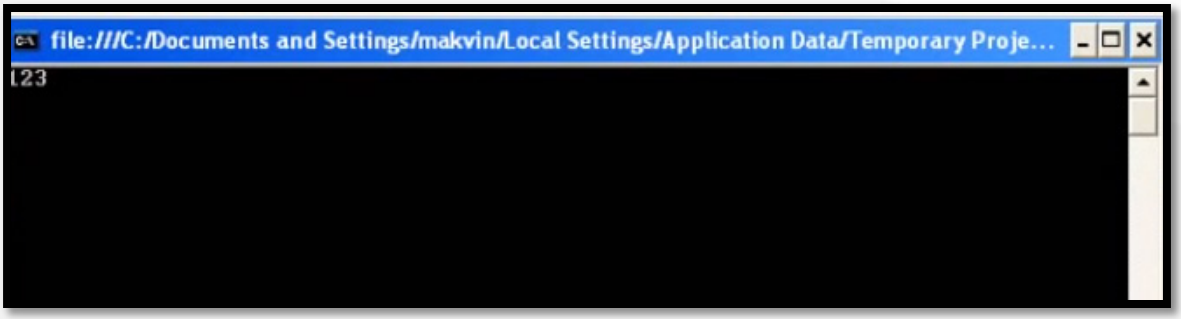
- Now it opens new window with Program.cs file name (i.e. Default page whenever we start console application).
- That program contain namespace as your file name (MathOperationsEg).
- In this page we need to start writing the program from main.



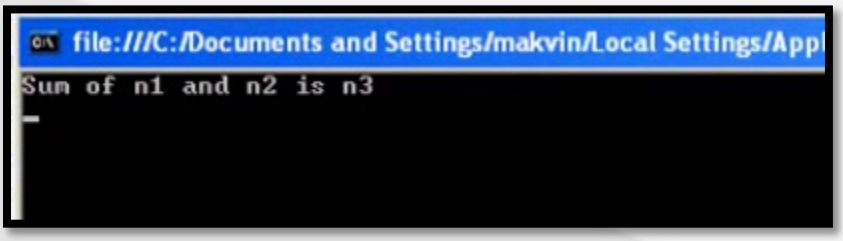
- We have three variables (n1, n2, n3) of integers.
- Our intention is to store 67 in n1, 56 in n2 and add n1 and n2 values and store in n3.
- Int n1,n2,n3;
- n1=67;
- n2=56;
- n3=n1+n2;
- We need to display the result as n3.
- To display the result we have method Console.WriteLine(n3);
- We need not to write n3 in quotation because we want to display the value of n3.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MathOperationsEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int n1, n2, n3;
            n1=67;
            n2=56;
            n3 = n1 + n2;
            Console.WriteLine(n3);
            Console.ReadLine();
        }
    }
}
```

- And ends with Console.ReadLine() method.
- To execute simply press F5.
- Result is 123 as shown below.



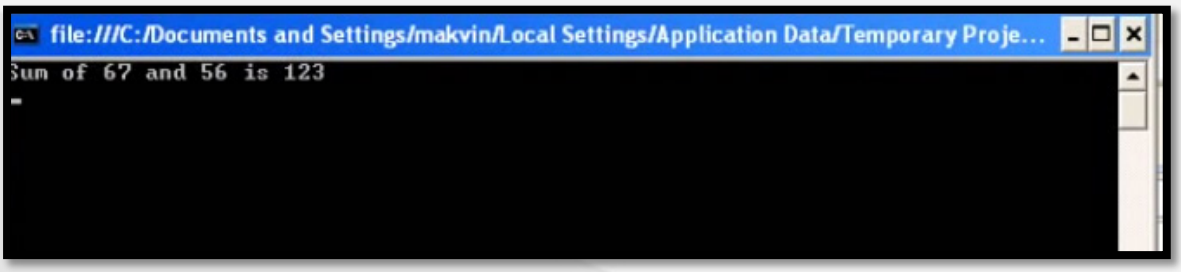
- Now we have to display the result as sum of n1 and n2 is n3.
- We want to display the result with meaningful message.
- We'll just put this in quotation inside the Console.WriteLine () method.
- Console.WriteLine("Sum of n1 and n2 is n3");
- Press F5.



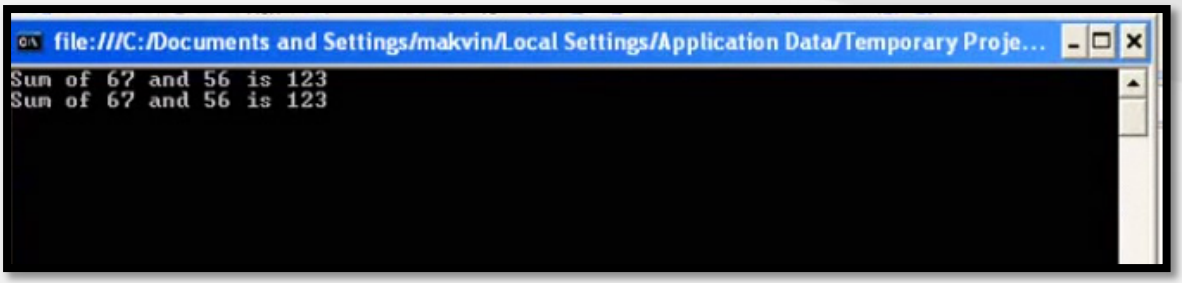
- Output is same as message inside the method.
- Instead of this we want to display the values of n1, n2 and n3.
- To display the values of n1, n2 and n3 simply we need to replace the n1, n2 and n3 with {0}, {1} and {2} as indexes in quotation.
- Now put comma after quotation and give the variable name at 0^{th} position (i.e. n1), n2 at 1^{st} position and n3 at 3^{rd} position.
- Cosole.WriteLine("sum of {0} and {1} is {2}",n1,n2,n3);

```
static void Main(string [] args)
{
    int n1, n2, n3;
    n1=67;
    n2=56;
    n3 = n1 + n2;
    Console.WriteLine( "Sum of {0} and {1} is {2}",n1,n2,n3);
    Console.ReadLine();
}
```

- Press F5.



- There is another way to display the same result using '+' operator.
- It will not work as mathematical operator but it works as string concatenation.
- Cosole.WriteLine("sum of "+n1+" and "+n2+" is "+n3);

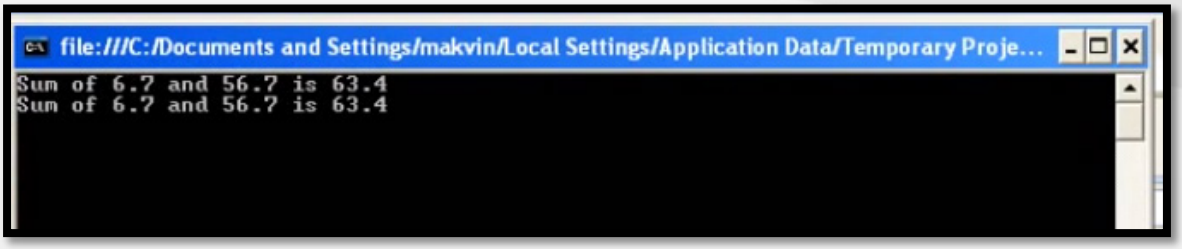

file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
Sum of 67 and 56 is 123
Sum of 67 and 56 is 123

- Both of the methods display same result but using two different techniques.
- One is passing parameters kind of things like giving indexes for n number of variables, and another is using '+' operator for string concatenation.

- Above program we get addition of two integer numbers if we give variable type as int.
- If we want to give some decimals to perform addition then we need to change the variable type 'int to double'.
- Double n1, n2, n3;

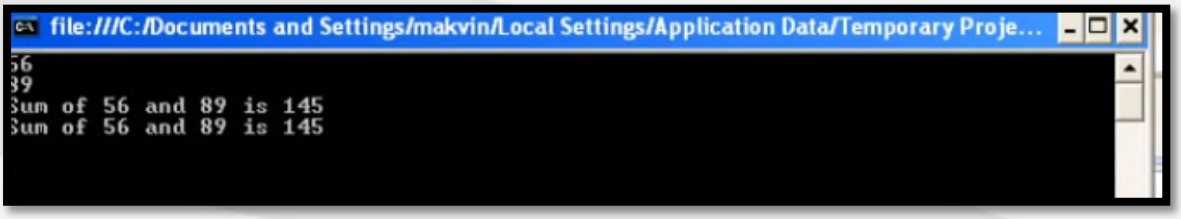
```
static void Main(string [] args)
    {
     Double n1, n2, n3;
     n1=6.7;
     n2=56.7;
     n3 = n1 + n2;
     Console.WriteLine("Sum of {0} and {1} is {2}",n1,n2,n3);
     Console.WriteLine("Sum of " + n1 + " and " + n2 + " is " + n3);

        Console.ReadLine();
    }
```

- Press F5.


file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
Sum of 6.7 and 56.7 is 63.4
Sum of 6.7 and 56.7 is 63.4

- If we want to add different values we need not to go and edit the program again and again.
- We can give the option to end user to enter the values from keyboard at runtime.

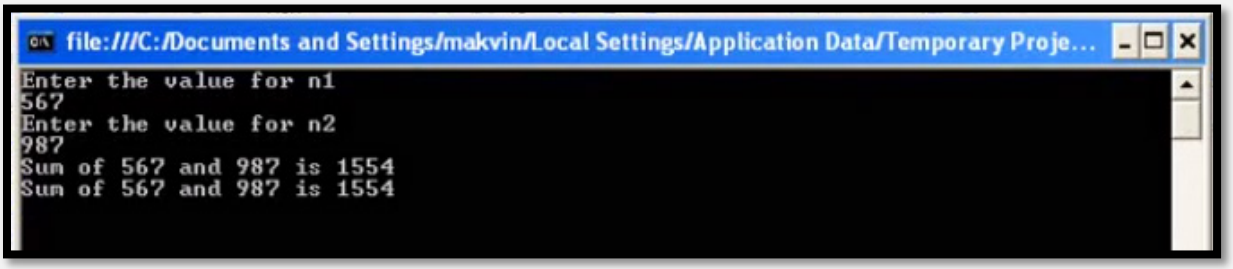
- For implementing this we can replace the values with Console.ReadLine() method means reading values from keyboard at runtime.
- But values read from keyboard at runtime is always in string format.
- And our variables (n1, n2) are in int or double type.
- So we need to convert this string type to integer type.
- We can achieve this by using int.Parse(Console.ReadLine()); (means parse this string to Int).
- n1= int.Parse(Console.ReadLine());
- n2= int.Parse(Console.ReadLine());
- If we execute this it will be waiting for taking two numbers.
- Enter the two numbers.
- Press enter for result.


file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
56
89
Sum of 56 and 89 is 145
Sum of 56 and 89 is 145

- For end users easy understandablility we can give messages before entering the values from keyboard.
- First message as “Enter the value for n1”.
- And second message as “Enter the value for n2”.

- Console.WriteLine(“Enter the value for n1”);
- n1= int.Parse(Console.ReadLine());
- Console.WriteLine(“Enter the value for n2”);
- n2= int.Parse(Console.ReadLine());

- Remaining part is same.
- Press F5.


file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
Enter the value for n1
567
Enter the value for n2
987
Sum of 567 and 987 is 1554
Sum of 567 and 987 is 1554

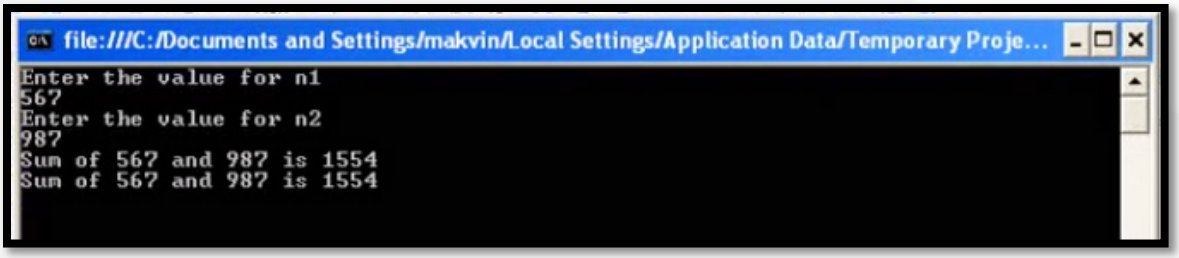
- If we have double values we need to change int type to double and instead of int.Parse we need to use double.Parse.

- Double n1,n2,n3;
- Console.WriteLine("Enter the value for n1");
- n1= double.Parse(Console.ReadLine());
- Console.WriteLine("Enter the value for n2");
- n2= double.Parse(Console.ReadLine());

- Complete code of addition of two integer numbers given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MathOperationsEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int n1, n2, n3;
            Console.WriteLine("Enter the value for n1");
            n1 =int.Parse(Console.ReadLine());

            Console.WriteLine("Enter the value for n2");
            n2 =int.Parse( Console.ReadLine());

            n3 = n1 + n2;
            Console.WriteLine("Sum of {0} and {1} is {2}",n1,n2,n3);
            Console.WriteLine("Sum of " + n1 + " and " + n2 + " is " + n3);

            Console.ReadLine();
        }
    }
}
```

- Press F5.

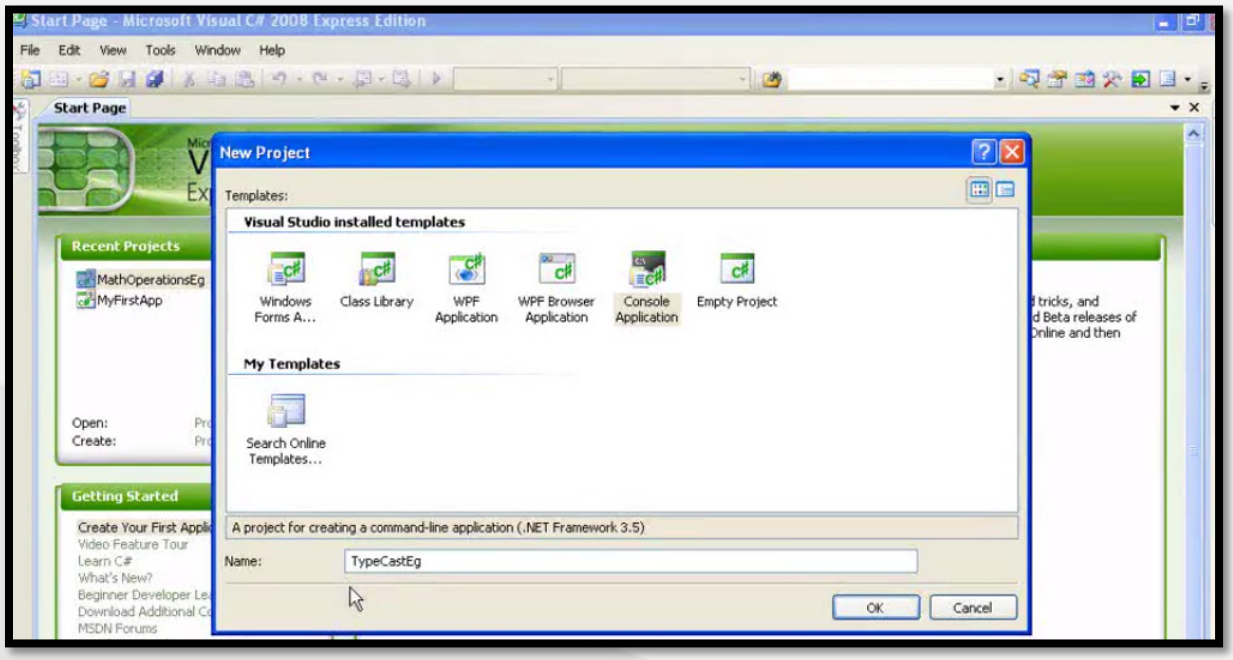

file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
Enter the value for n1
567
Enter the value for n2
987
Sum of 567 and 987 is 1554
Sum of 567 and 987 is 1554

- In this chapter we have seen addition of two integers and double numbers.
- Display methods using various techniques.
- Reading values from keyboard.
- Int.Parse and double.Parse for conversion.

Thank you..!

CHAPTER 2: TYPE CASTING

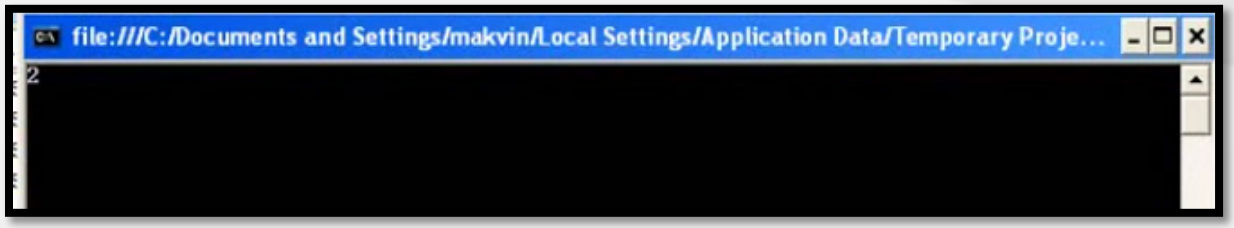
- In this chapter we'll see the concept of type casting.



- Select new project→console application
- Name it as TypeCastEg click on ok.
- For example we have two integer variable a and b.
- Int a=5;
- Int b=2;
- If we say console.writeline(a/b), could you guess the output?.
- Definitely we expect the output to be 2.5 as per our mathematical operations.
- Let us see what it says the result here.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TypeCastEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int a = 5;
            int b = 2;

            Console.WriteLine(a/b);
            Console.ReadLine();
        }
    }
}
```

- We'll just press F5.



- It says that result is 2.
- Because if both variables are integer our result also be an integer.
- If we observe this thing.

- 5+2=7.
- If we say 5.0+2=7.0.
- See that there is not much difference in mathematical concept 7 and 7.0 both are same.
- Whereas in our computer world 7 and 7.0 has lot of difference in their data structure.
- If we say 5+2.0 definitely our result would be 7.0.
- If we say 5.0+2.0 our result would be again same as 7.0.
- One thing we need to observe here is '5' is our first operand '2' is our second operand and '+' called as operator.
- If our both the operands are integer our result is integer.
- One of the two operands is double then our result is double.
- If anyone operand is double our result will be double.
- In the same way we are performing 5/2 the result will be 2.
- Why because '5' is an integer '2' is an integer, so integer/integer will gives rise to integer?
- We want the result to be double what is that we need to do, we need to make either 5 as double or 2 as double or both of them as double.
- We'll make 5 as double.
- 5.0/2 this will give the result as 2.5.

- Now the same thing we need to implement in program.

```
Console.WriteLine(5.0/2);
Console.ReadLine();
```

- We got 2.5.
- Now what we want we've 'a/b', now how do we make this as in points.
- If we make int as double.

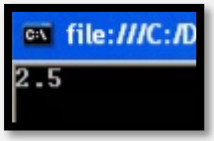
```
double a = 5.0;
int b = 2;

Console.WriteLine(a/b);
Console.ReadLine();
```

- This could be one of the solution but this is not correct way.
- Why because only once we want to perform division but maybe we know n number of times we want to perform addition, subtraction and multiplication, so in that cases if we want to add two integer it takes very very less amount of time when we compare it with adding an integer and a double.
- Adding an integer with double takes huge amount of time whereas it takes less amount of time to add two integers.
- So may be in our program we need only once for division may be n number of times we may be adding it.
- So addition of two integers takes very very less amount of time when compare to an addition of integer and a double.
- We don't want to change the data type we want it to be an integer.
- But at the time of performing division operation we want to make this as double.

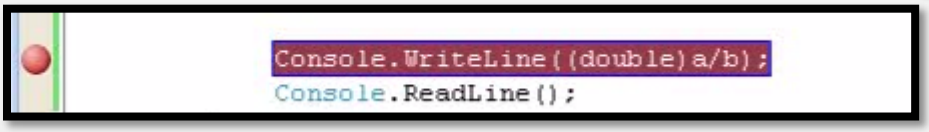
- We can do this by making ‘a’ as double only at the time of division.
- We need to write ‘(double)’ before ‘a’.

```
Console.WriteLine((double)a/b);
Console.ReadLine();
```

- Now we’ll execute it.

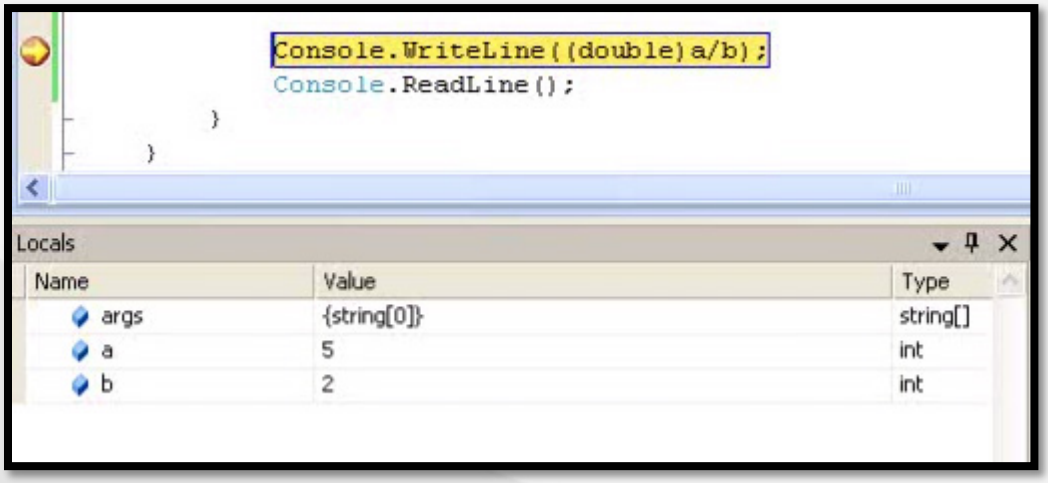


- We got the result.
- What it that’s perform.
- It’ll take the value of ‘a’ as ‘5’ and it will convert this to ‘double’ only at the time of division operation.
- That means it’s not changing the data structure from integer to double permanently.
- Temporarily it is casting it from integer to double.
- So this is called as ‘type casting’.
- If we want we can put a ‘break point’.
- What is ‘break point’, on the left displaying area we’ll just click over here?

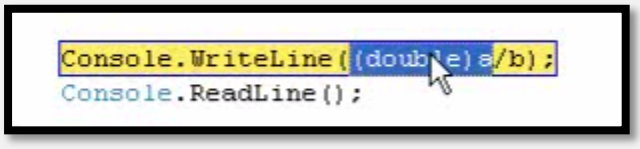


- This line gets highlighted.
- If we click it again the red boll gets disappeared if we click its once again we got red boll.
- We call it as ‘break point’.

- Our program execution will come till this point and it will stop.
- From there we want to see the execution we can just press F11 to see the execution one step after the other.
- Or we can just press F5 to see the complete result or to ignore this break point.
- Now we'll just press F5.



- Now our program execution stops here.
- We can see that yellow bar that means our control is here.
- We can see the value of 'a or b' we just take the mouse pointer on a or b.
- If we select double of 'a' our value will be 5.0.

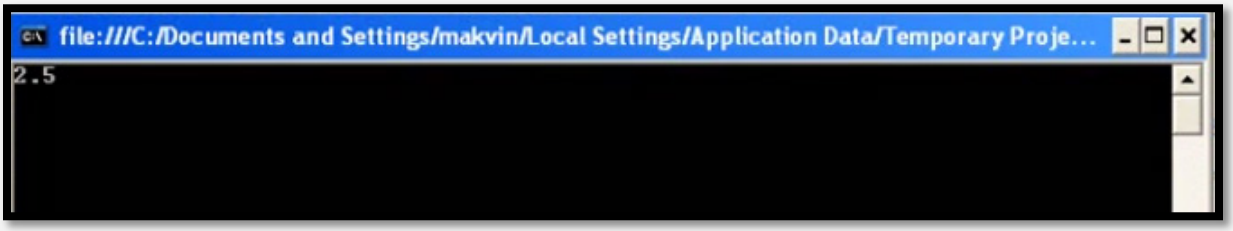


- That means we've converted or we've type casted the value of 'a' from integer to double.
- For example if we've 5 billion dollars of business.
- We want to divide the profit in two equal shares to share holder A and shareholder B.

- If we do not type cast of this things then we'll be simply losing 0.5 million dollars, so that is not a small amount.
- So what is that we need to do we need to go for type casting.
- Concept is very small but very important.
- See the code given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TypeCastEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int a = 5;
            int b = 2;

            Console.WriteLine((double)a/b);
            Console.ReadLine();
        }
    }
}
```

- Above code casts 5 to 5.0.
- And perform division operation (5.0/2) and print the result as 2.5.
- Press F5.



- Now our result is as expected.

CHAPTER 3: CONTROL STRUCTURES

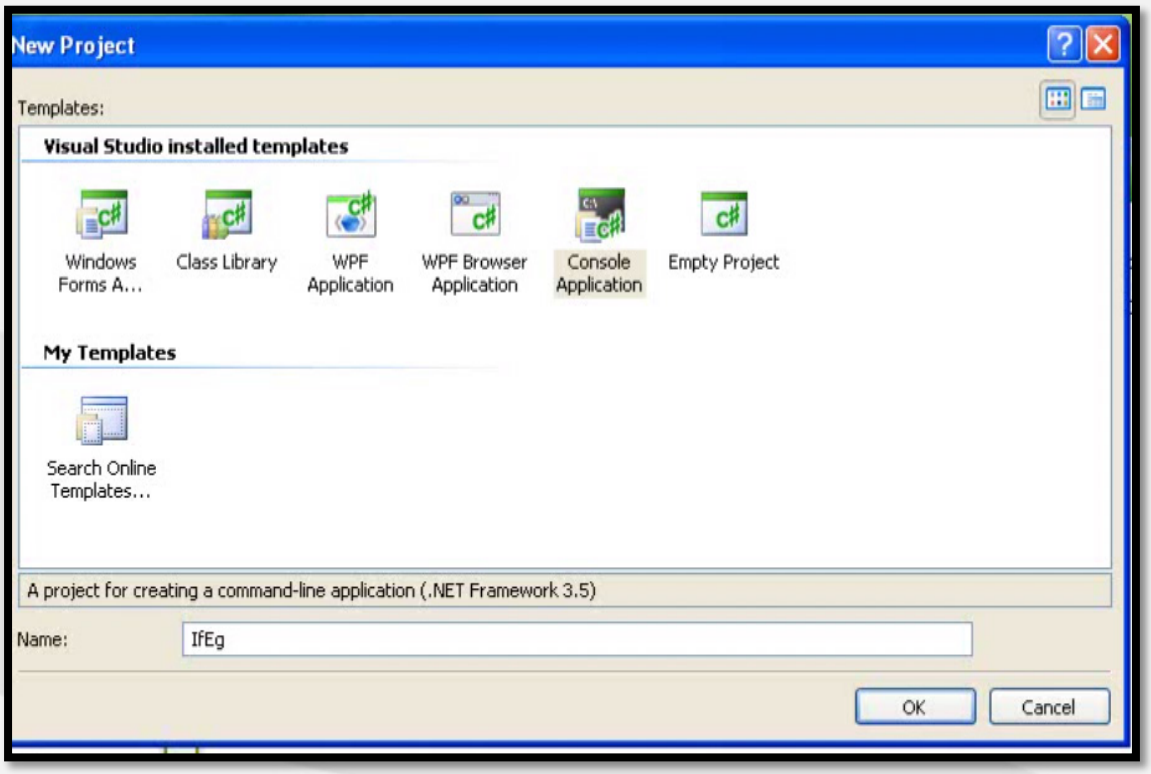
- In this chapter we'll proceeding towards control structures.
- Control structures controls the flow of program execution
- We've three types of control structures.

```
Control Structure

        1.Conditional

                a.if
                b.if else
                c.else if Ladder
                d.Switch Case Default


        2.Unconditional

                a.break
                b.continue
                c.goto
                d.return


        3.Iterative or Loops

                a.for
                b.while
                c.do while
                d.foreach(We will be dealing this after Arrays)
```

1. CONDITIONAL:

IF AND IF-ELSE CONDITION:

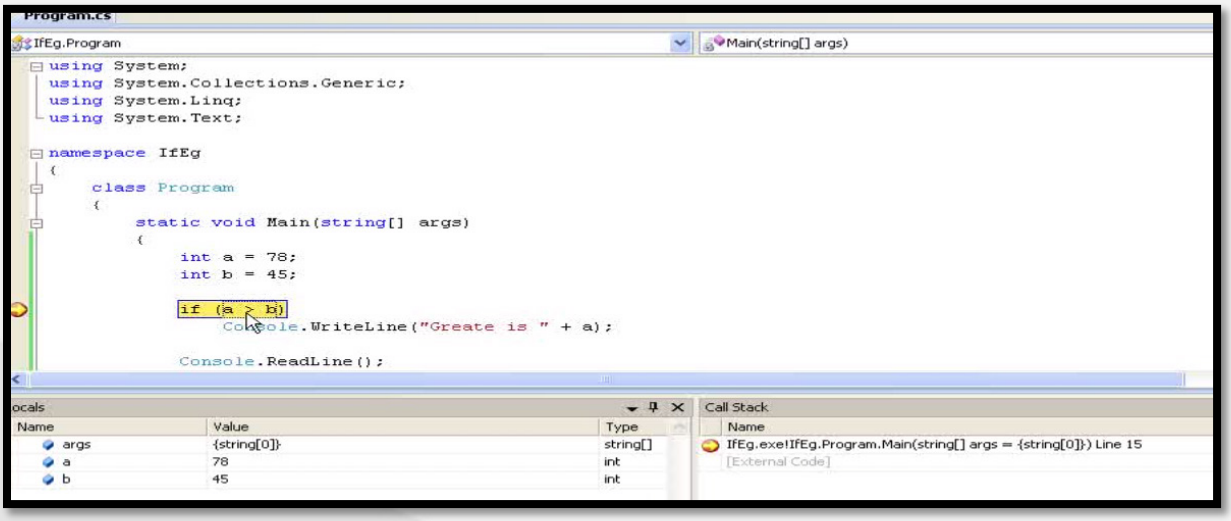
- Start visual studio choose new project.
- And give it a name.



- Click on ok.
- In this program we'll check greater number between two variables.
- We have declared two variable with some values.
- And we'll check greater number, for this we need to write if (a>b) condition.

```
int a = 78;
int b = 45;

if (a > b)
    Console.WriteLine("Greate is " + a);

Console.ReadLine();
```

- If we observe if ‘a’ is greater than ‘b’ then we print the statement as “Greater is “+a.
- And we’ll put a break point over if condition then press F5.
- And then press F11 to see the program execution line by line.

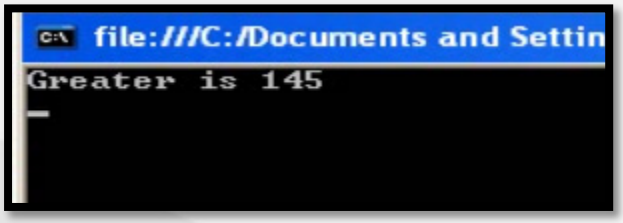


- In above snap we can see that value of a is 78, and value of b is 45 then a greater than b condition become true.
- If we press F11 it will going to execute ‘Cosole.WriteLine()’ statement.
- Now statement “Greater is 78” will be displayed.

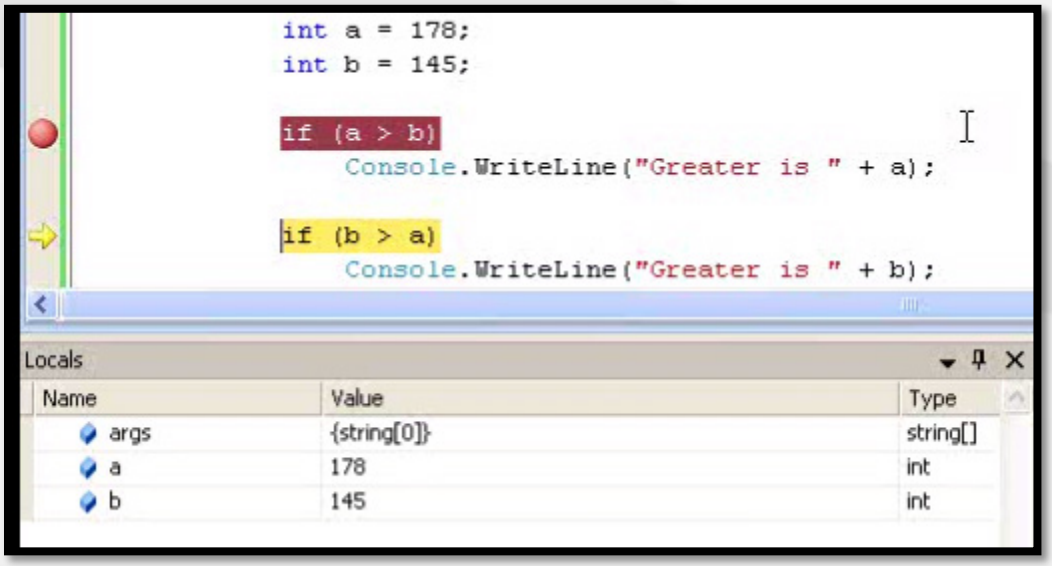
- If we change the value of b to 145.
- We need to write one more if case as here

```
static void Main(string[] args)
{
    int a = 78;
    int b = 145;

    if (a > b)
        Console.WriteLine("Greater is " + a);

    if (b > a)
        Console.WriteLine("Greater is " + b);

    Console.ReadLine();
}
```

- Now our output will be



- If we put a break point.

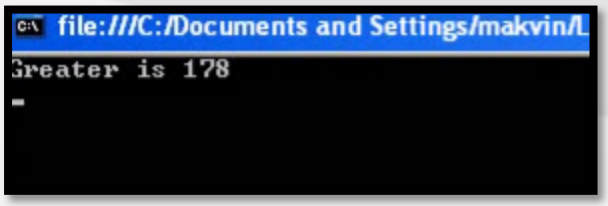


- Here it checks both the conditions.
- If a>b definitely b will not be greater than a.

- So its skip this.
- Using if we do not have that option.
- It needs to check first condition then comes out of the program.
- We want to skip second condition.
- Instead of comparing b>a condition we can simply replace second condition with **else.**

```
static void Main(string[] args)
{
    int a = 178;
    int b = 145;

    if (a > b)
        Console.WriteLine("Greater is " + a);

    else
        Console.WriteLine("Greater is " + b);

    Console.ReadLine();
}
```

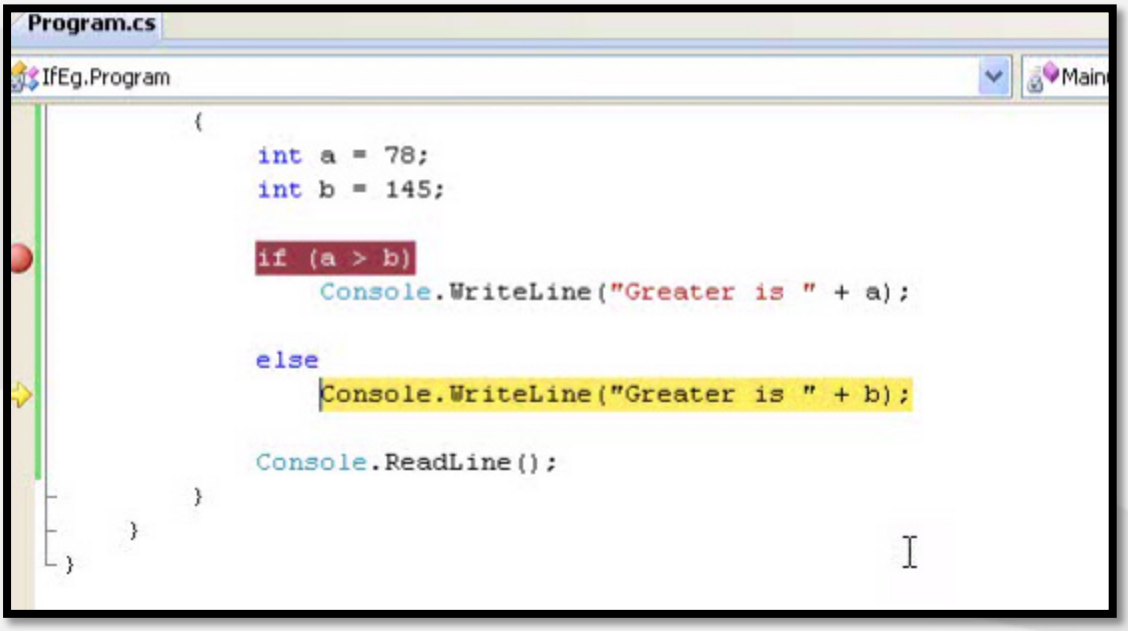
- First it will check first condition a>b if it's true then print the result.
- And skip the else part and then comes out of the block



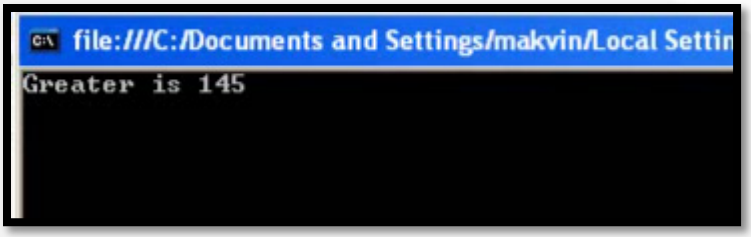
- Check in another way.

```
static void Main(string[] args)
{
    int a = 78;
    int b = 145;

    if (a > b)
        Console.WriteLine("Greater is " + a);

    else
        Console.WriteLine("Greater is " + b);

    Console.ReadLine();
}
```

- Now put a break point check the execution press F5.



- Now we can see that condition a>b is false.
- So its skip the if part and jumps to else part.
- Then prints the result as “Greater is 145”
- Press F5.



- As our expected result 145 is shown.
- So this is our simple greater of two numbers using if-else condition.

ELSE-IF LADDER:

- In this chapter we'll see greatest of three numbers using else-if ladder.
- Choose another console application.
- To find greatest of three numbers we need three variables.

```
int a = 34;
int b = 78;
int c = 67;
```

- Now start checking conditions. a>b and a>c.
- To perform two conditions in single if, we need to use 'logical and' operator (&&).

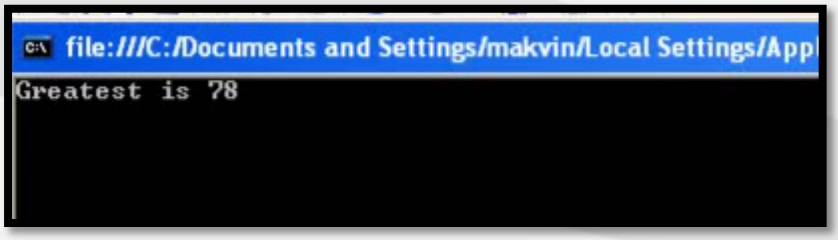
```
if (a > b && a > c)
    Console.WriteLine("Greatest is " + a);
```

- Else then we need to nest we should go for nesting.
- We've checked for if then in else part we need to nest.
- In our else part we'll write another if-else.
- This called as nested if-else.
- Now in else block check the condition for greatest of b and c.

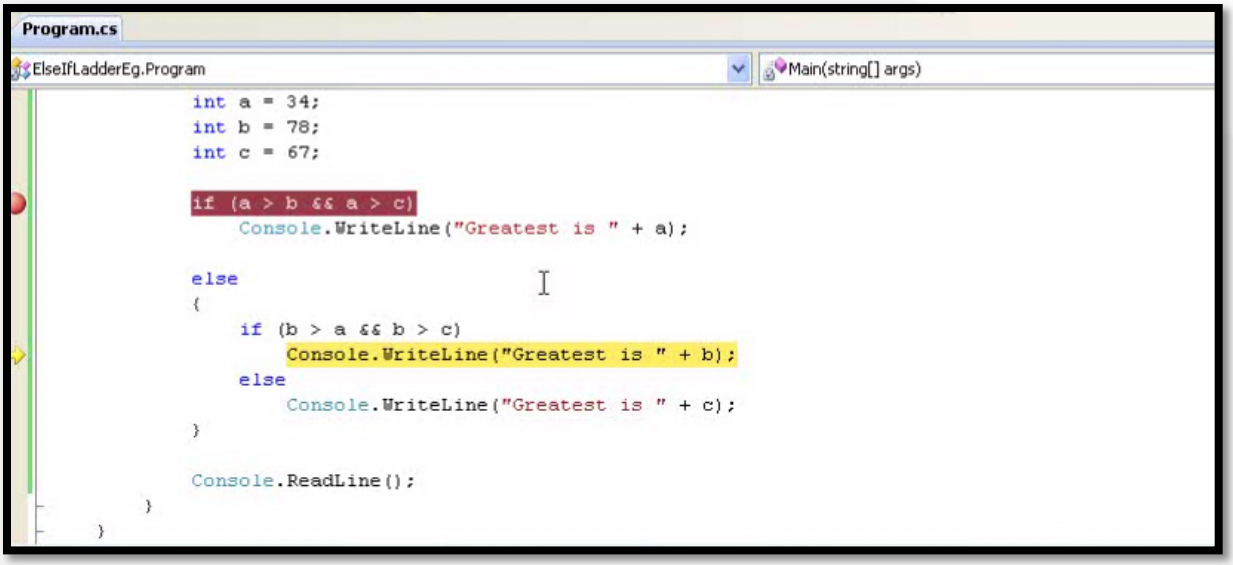
- So open the brackets in else block (if we have multiple statements in if or else block use brackets).

```
static void Main(string[] args)
{
    int a = 34;
    int b = 78;
    int c = 67;

    if (a > b && a > c)
        Console.WriteLine("Greatest is " + a);

    else
    {
        if (b > a && b > c)
            Console.WriteLine("Greatest is " + b);
        else
            Console.WriteLine("Greatest is " + c);
    }

    Console.ReadLine();
}
```

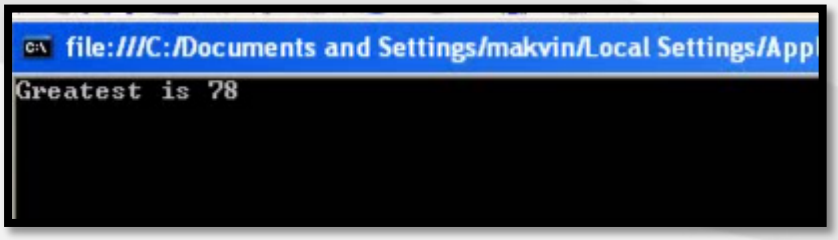
- Press F5.



- Put a break point see the flow of execution.



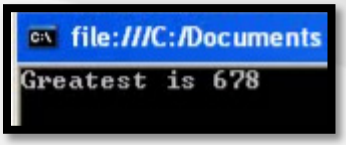
- First it will check if condition a>b and a>c, here both are false so its skip the if block and enter into else block.
- In else block we've again if condition, it checks the condition for if (b>a && b>c) here both parts are true so whole condition becomes true, so control enters into if block and executes the statement and comes out of the block.
- It need not to go for the else part.



- So the greatest is 78.
- Till this it's fine but our program complexity gets increased when we want to find out the greatest of four numbers.

```
static void Main(string[] args)
{
    int a = 34;
    int b = 78;
    int c = 67;
    int d = 678;

    if (a > b && a > c && a>d)
        Console.WriteLine("Greatest is " + a);

    else
    {
        if (b > a && b > c && b > d)
            Console.WriteLine("Greatest is " + b);
        else
        {
            if (c > a && c > b && c > d)
                Console.WriteLine("Greatest is " + c);
            else
                Console.WriteLine("Greatest is " + d);
        }
    }

    Console.ReadLine();
}
```

- Here we are nesting else part.
- Check the execution.



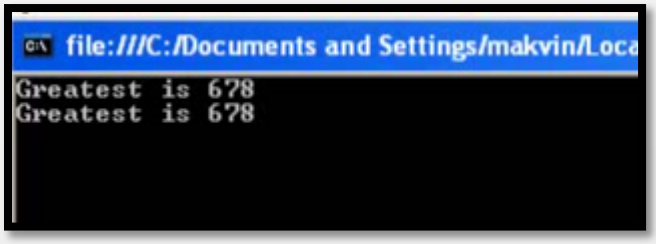
- Its working fine but logic become more complex and it gives tough readability.
- In above program we've used many nested if-else.
- As many nested if-else becomes more complex to understand.
- To overcome this issue we have one solution i.e. **else-if ladder.**
- It's very simple.

```
if (a > b && a > c && a > d)
    Console.WriteLine("Greatest is " + a);
else if (b > a && b > c && b > d)
    Console.WriteLine("Greatest is " + b);
else if (c > a && c > b && c > d)
    Console.WriteLine("Greatest is " + c);
else
    Console.WriteLine("Greatest is " + d);
```

- Above code works same as earlier code but it gets easier than earlier to write as well as to understand.
- In the above situation it checks first if block, if all conditions in this block become true then it skips remaining block.
- Put a break point check the execution for above code.

```
if (a > b && a > c && a > d)
    Console.WriteLine("Greatest is " + a);
else if (b > a && b > c && b > d)
    Console.WriteLine("Greatest is " + b);
else if (c > a && c > b && c > d)
    Console.WriteLine("Greatest is " + c);
else
    Console.WriteLine("Greatest is " + d);
```

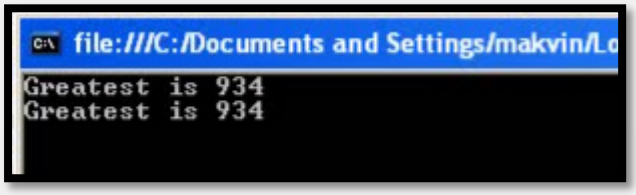
- Execution skip the first three blocks due to false condition.
- And jumps to else part.
- Now prints the result as greatest is 678.



- If a=934 then first condition block become true and skip all remaining else-if ladder block.
- Now print the result as Greatest is 934.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ElseIfLadderEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int a = 934;
            int b = 78;
            int c = 67;
            int d = 678;

            if (a > b && a > c && a>d)
                Console.WriteLine("Greatest is " + a);

            else
            {
                if (b > a && b > c && b > d)
                    Console.WriteLine("Greatest is " + b);
```

```
            else
            {
                if (c > a && c > b && c > d)
                    Console.WriteLine("Greatest is " + c);
                else
                    Console.WriteLine("Greatest is " + d);
            }
        }



        if (a > b && a > c && a > d)
            Console.WriteLine("Greatest is " + a);
        else if (b > a && b > c && b > d)
            Console.WriteLine("Greatest is " + b);
        else if (c > a && c > b && c > d)
            Console.WriteLine("Greatest is " + c);
        else
            Console.WriteLine("Greatest is " + d);

        Console.ReadLine();
    }
  }
}
```



SWITCH CASE:

- In this chapter we are going to write a program which accepts a value from keyboard from 1-4 and depending upon the input value it is going to perform some mathematical operations.
- If input value is 1-add two integers.
- If input is 2- subtract two integers.
- If input value is 3-for multiply.
- If input value is 4-divide two integers.
- These options should be enabled for the end-user to enter the choice.
- This can be achieve by using switch case.
- When should we go for switch case, we'll go for switch case whenever we've some multiple options we need to select one then we should go with switch cases.
- Same thing we might have observe when we have multiple options once we go for swiping our ATM card in ATM machine.
- It asks for saving account or current account.
- If we say saving account.
- Then it asks for the operations do you want to perform.
- Do you want to deposit amount or withdraw amount or balance enquiry.
- We have multiple things and we need to select one thing.
- So that kind of functionality whenever we want to implement we should go with switch cases.
- Here we need to display the options first.
- And read the choice.

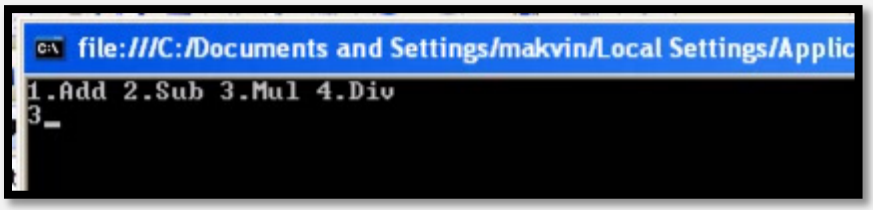
```
int a = 45;
int b = 34;

Console.WriteLine("1.Add 2.Sub 3.Mul 4.Div");
int ch =int.Parse(Console.ReadLine());
```

- We are reading user choice in 'ch' variable and performing four cases as below

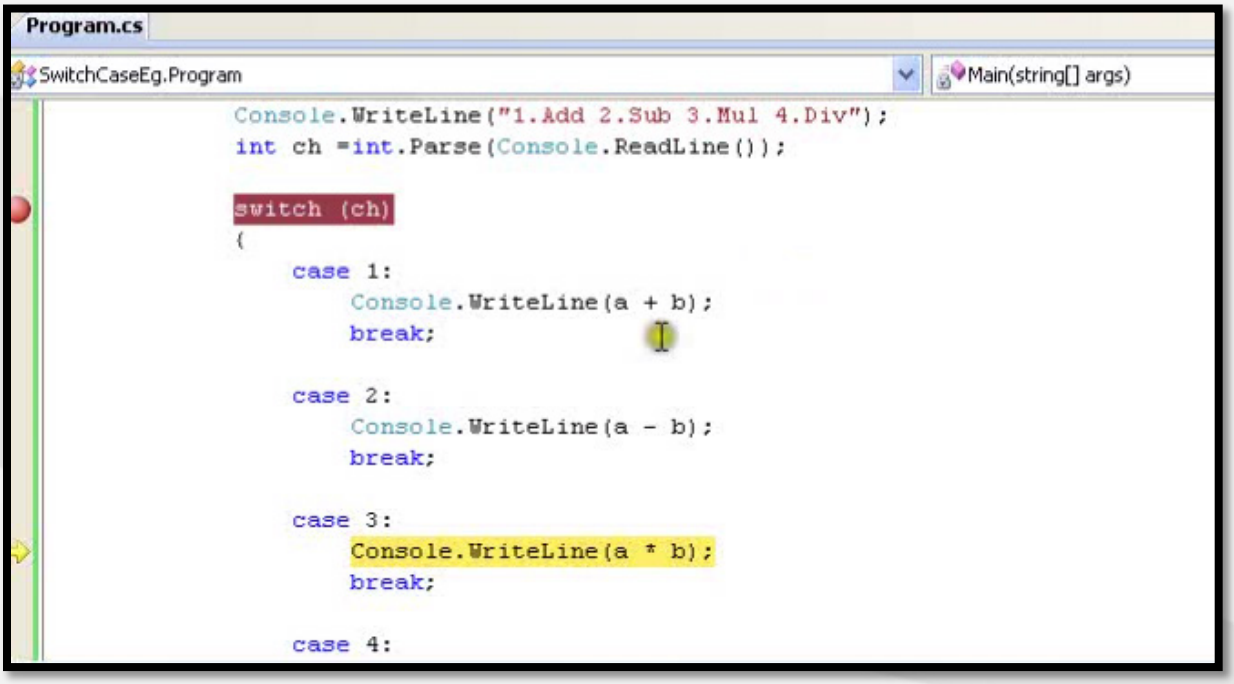
```
switch (ch)
{
    case 1:
        Console.WriteLine(a + b);
        break;

    case 2:
        Console.WriteLine(a - b);
        break;

    case 3:
        Console.WriteLine(a * b);
        break;

    case 4:
        Console.WriteLine((double)a / b);
        break;

    default:
        Console.WriteLine("Invalid Input");
        break;
}
```

- Based on **switch (ch)** value we are entering into the cases. ('ch' is user choice).
- We are giving user option in 'ch'.
- If it is 1 then it will jump into the case 1 and execute addition operation.
- And it will break.
- As soon as it encounters the break statement i.e. unconditional statement, without looking for any kind of condition it will directly jump out of the block.

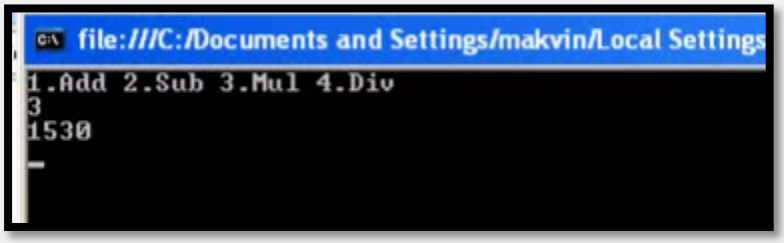
- We have one **default** case, if choice of the user is other than these cases then it enters into default case.
- Let us see the execution, put a break point and press F5.



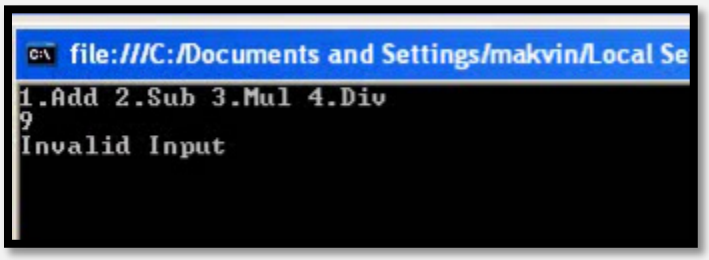
- User choice is 3.



- Now it checks the case i.e. 3
- It enters into the case 3: perform multiplication.
- And prints the result.



- If user enters the value other than these choices then it jumps into default case and prints “Invalid Input”.

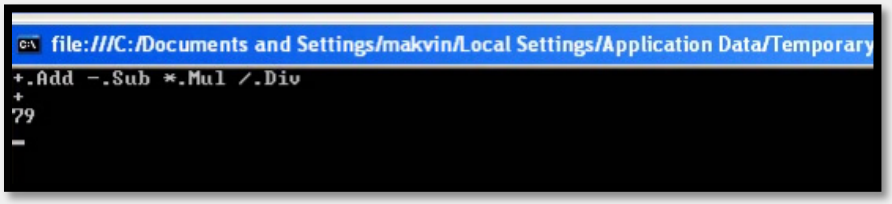


- One more thing this cases need not be in order we can change the order of cases also.
- We can start with case 3 and we can end it with case 2.
- We can give special characters as options in switch but we have to give same characters in double quotation as cases.
- And complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SwitchCaseEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int a = 45;
            int b = 34;
            Console.WriteLine("+.Add -.Sub *.Mul /.Div");
            string ch =Console.ReadLine();
```

```
            switch (ch)
            {
                case "*":
                    Console.WriteLine(a * b);
                    break;

                case "/":
                    Console.WriteLine((double)a / b);
                    break;
                case "+":
                    Console.WriteLine(a + b);
                    break;
                case "-":
                    Console.WriteLine(a - b);
                    break;

                default:
                    Console.WriteLine("Invalid Input");
                    break;
            }
            Console.ReadLine();
        }
    }
}
```

- Press F5.

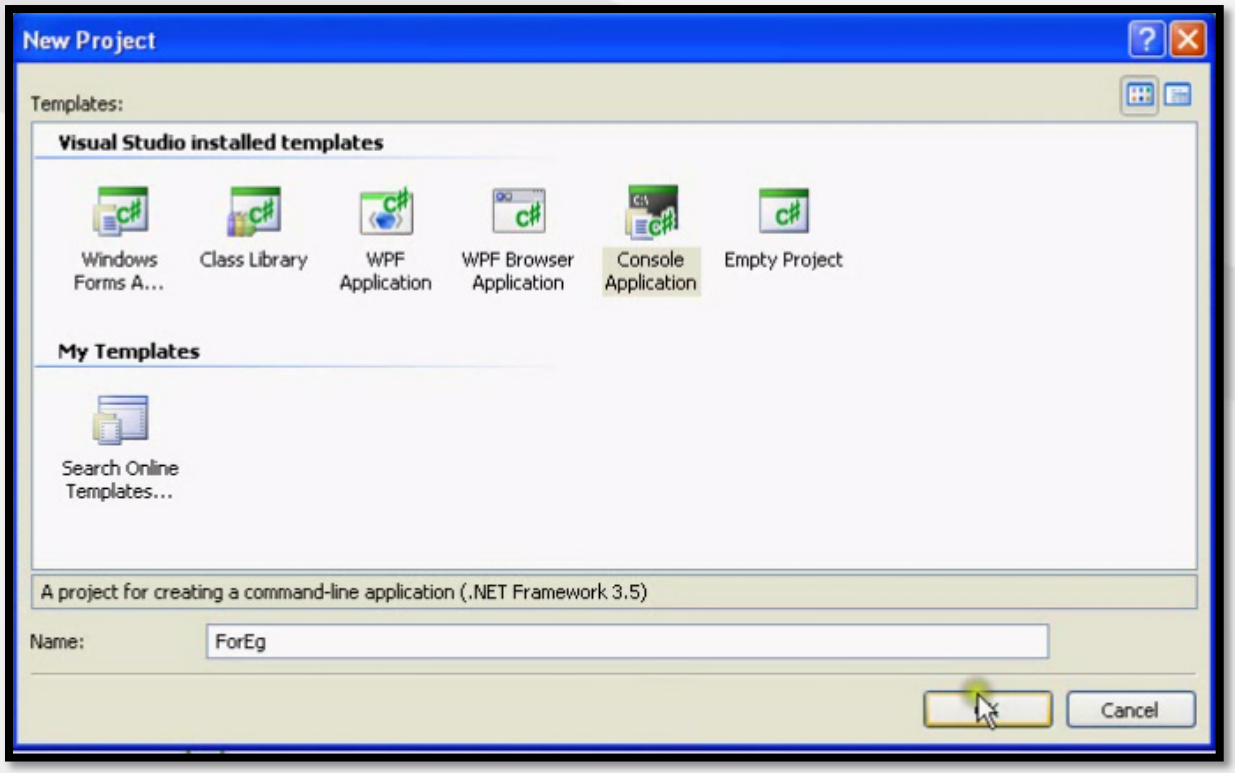


2. ITERATIVE STATEMENTS OR LOOPS:

- In this chapter we'll see iterative statements or we can call it as looping structures.
- In our control structure we've seen conditional statements now we'll proceed towards iterative statements.
- There are various iterative statements that we have.
- For loop, while loop, do while and foreach.

FOR LOOP:

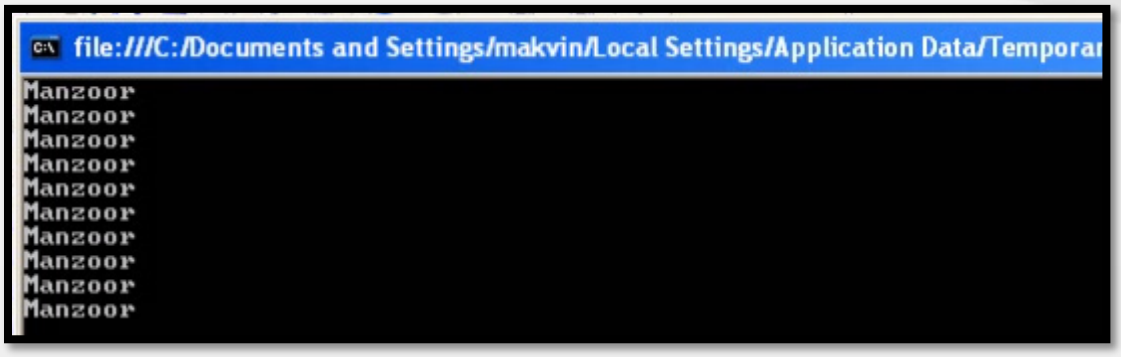
- Let us start new project where we will write sample program to explain for loop.
- Program→ Visual studio→File→New project→Console application.
- Name it as 'ForEg'.



- In this section we'll write the program to display the name for 10 times using for loop.
- For loop is used for definite iteration, which means we know the number of iterations prior to execution.
- For example
- for(int i=0;i<10;i++)
- {
- }
- It will start from 0 and executes whatever we write in this block for 10 times.
- From 0 till 9, 'i' value will vary from 0 till 9.
- Now we'll display message for 10 times.

```
static void Main(string[] args)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine("Manzoor");
    }
    Console.ReadLine();
}
```

- Press F5.



- So it displays the name for 10 times.

- Let us see how it works.
- First control moves to initialization part

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Manzoor");
}
```

- It initializes the variable i to 0.
- Next checks the condition.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Manzoor");
}
```

- 0 is less than 10, so condition is true.
- It will enter into loop to print the statement.
- After printing, control moves to increment/decrement.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Manzoor");
}
```

- So i value gets incremented to 1.
- Now control again moves to check the condition.

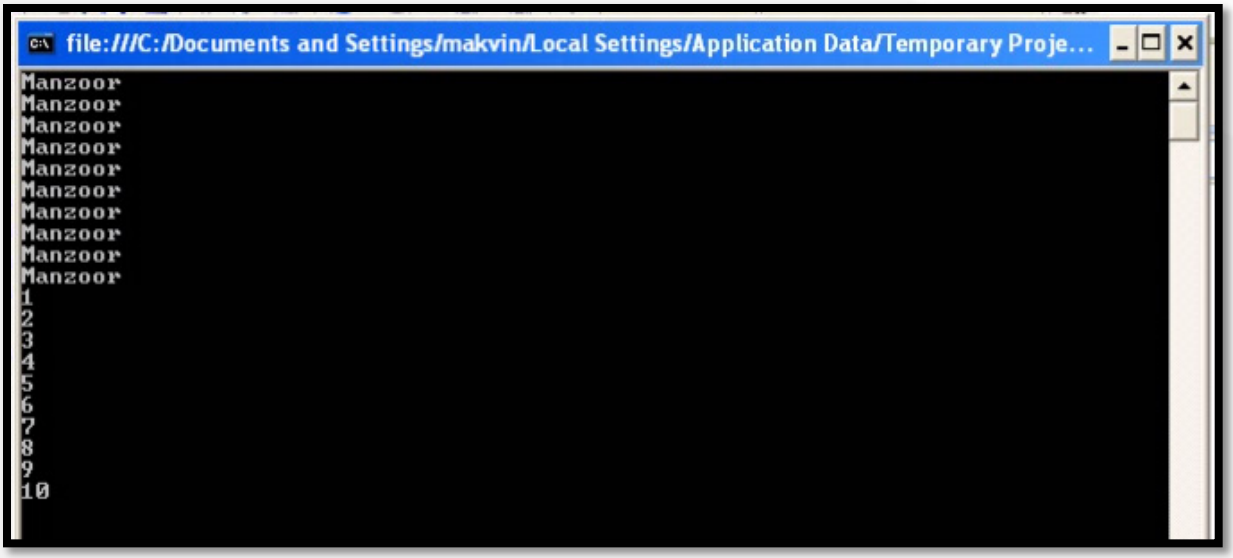
```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Manzoor");
}
```

- If condition satisfies, then control moves to print the statement again.
- If I value becomes 10 now condition ‘i<10’ becomes false.
- So it will skip the loop and comes out of the loop

- Means it repeats the process until condition becomes false.
- If condition becomes false then control jumps out of the loop.
- This is simple for loop which displays name for 10 times.
- Now we want to display integers from 1 to 10.
- Same we can use for loop.
- We have code snippet for 'for loop', type 'for' and press tab twice to generate for loop readymade code.

```
static void Main(string[] args)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine("Manzoor");
    }

    for (int i = 1; i <=10; i++)
    {
        Console.WriteLine(i);
    }

    Console.ReadLine();
}
```

- In earlier code we've displayed name, whereas here we are trying to display the value of 'i'.
- it will display all integers from 1 to 10.
- Press F5.



- Now we'll print numbers from 10 to 1.

```
for (int i = 10; i >= 1; i--)
{
    Console.WriteLine(i);
}
```

- Here i value is initialized to 10, check the condition for i >=1 means till 1, and use decrement operator.

- Now i value gets decremented from 10 to 1.
- Here is the code for displaying even numbers from 1 to 100.

```
for (int i = 1; i <= 100; i++)
{
    if (i % 2 == 0)
        Console.WriteLine(i);
}
```

- Code for displaying odd numbers.
- It displays all odd numbers from 1 to 100.

```
for (int i = 1; i <= 100; i++)
{
    if (i % 2 == 1)
        Console.WriteLine(i);
}
```

- Now we'll see the code for multiplication table of 6.

```
int n = 6;

for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("{0} * {1} = {2}", n, i, n * i);
}
```

- Table format is given in above Cosole.WriteLine() method.
- Value of 'n' comes in zeroth position, value of 'i' comes in first position, and value of 'n*i' comes in second position.
- This will print the multiplication table of number 6 from 1 till 10.
- Press F5.

```
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
6 * 10 = 60
```

- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ForEg
{
    class Program
    {
        static void Main(string [] args)
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine( "Manzoor");
            }

            for (int i = 1; i <=10; i++)
            {
                Console.WriteLine(i);
            }

            for (int i = 10; i >= 1; i--)
            {
                Console.WriteLine(i);
            }

            for (int i = 1; i <= 100; i++)
            {
```

```
            if (i % 2 == 1)
                Console.WriteLine(i);
        }

        int n = 6;

        for (int i = 1; i <= 10; i++)
        {
            Console.WriteLine("{0} * {1} = {2}", n, i, n * i);
        }

        Console.ReadLine();
    }
  }
}
```

- This is how our for loop works.

WHILE LOOP:

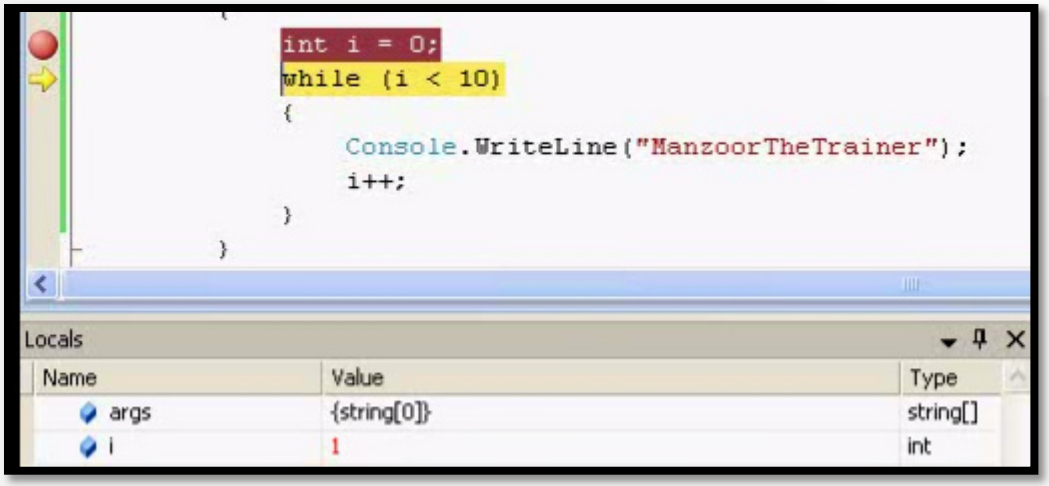
- Program→ Visual studio→File→New project→Console application.
- Name it as 'WhileEg'.
- While loop is used whenever we have the situation of indefinite iterations.
- Means number of iterations are unknown prior to execution we go for **while** loop
- Here is the simple code for understand while loop.

```
static void Main(string[] args)
{
    int i = 0;
    while (i < 10)
    {
        Console.WriteLine("ManzoorTheTrainer");
        i++;
    }
}
```

- It looks similar to our 'for loop' but to understand we are using while loop to implement this task.
- Because if number of iterations are known prior to execution we should go with 'for loop'.
- While loop is specially designed for those iterations which are unknown prior to execution.
- We'll put a break point it very simple and straight forward.

```
int i = 0;
while (i < 10)
{
    Console.WriteLine("ManzoorTheTrainer");
    i++;
}
```

- First I value will be 0 then it checks the condition i.e. 0<10 condition becomes true.
- It will executes the statement and I value gets incremented.
- Again it jumps to the condition.



- Here checks the condition i.e. 1<10 condition true.
- It will execute the statement.
- Then again increment the value of 'I'.
- Now 'I' value become 2.
- Again it jumps to the condition.
- And this happens as long as the 'i' value remain less than 10.
- So it is going to execute it for the 10 times.
- Here one example to understand while loop.
- We want to display the sum of all the digits using while loop.
- Logic is we'll take the digits one after the other from right side by applying modulus.
- That is we'll try to find out the reminder by dividing it by 10 then we'll get the last digit.
- Then we'll shortened this number by dividing it by 10 which will be the quotient.
- We'll use the simple logic of reminder and quotient to implement this.
- Below code is to implement the sum of all the digits of a given number.

```
int n = 5674;
int sum = 0;
int r;
while (n != 0)
{
    r = n % 10;
    sum = sum + r;
    n = n / 10;
}

Console.WriteLine("Sum is " + sum);

Console.ReadLine();
```

```
Sum is 22
```

- Our sum is 22.
- If we read the value from keyboard.

```
int n = int.Parse(Console.ReadLine());
int sum = 0;
int r;
while (n != 0)
{
    r = n % 10;
    sum = sum + r;
    n = n / 10;
}

Console.WriteLine("Sum is " + sum);
```

- Now press F5.

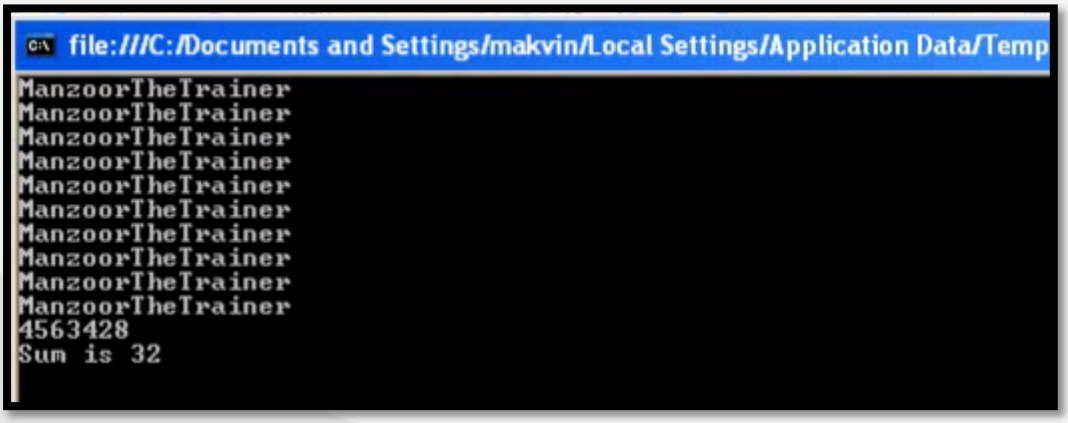
```
4563428
Sum is 32
```

- In above case while loop is used because number of iterations are unknown as a number can have n number of digits.
- Complete code is given for sum of the digits is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace WhileEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int i = 0;
            while (i < 10)
            {
                Console.WriteLine( "ManzoorTheTrainer" );
                i++;
            }

            int n = int.Parse(Console.ReadLine());
            int sum = 0;
            int r;
            while (n != 0)
            {
                r = n % 10;
                sum = sum + r;
                n = n / 10;
            }

            Console.WriteLine( "Sum is " + sum);
```

```
            Console.ReadLine();
        }
    }
}
```

- Press F5.



- User entered 4563428
- Sum of the digits is 32.
- So this is all about while loop its syntax, how it works and when we are going to use while loop.

DO-WHILE LOOP:

- Do-while loop is used again, whenever we have indefinite number of iterations.
- But difference between while and do-while is, while checks the condition first then execute the statements whereas do-while it executes the statements first and then it goes for checking the condition.
- We can use do-while loop when iterations are unknown and need to executes the task at least once.
- We've unknown number of iterations and we need to execute the task at least once in that kind of situations we should go for do-while loop.

- For example we want to write a program where we need to accept the numbers from the keyboard unless and until we get zero.
- That means as long as we getting a non-zero value we need to accept the value from the keyboard.
- That means these are unknown number of iterations and we need to execute it once.
- So at least one digit we should have whether it is zero or non-zero to check.
- Syntax is
- Do
- {
- }while(condition);

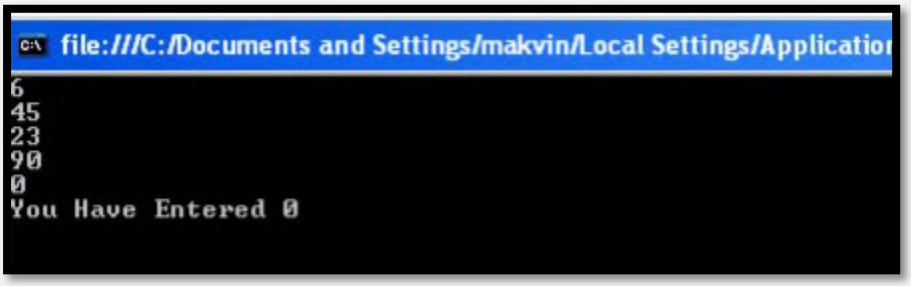
```
static void Main(string[] args)
{
    int n;
    do{
        n=int.Parse(Console.ReadLine());
    }
    while(n!=0);

    Console.WriteLine("You Have Entered " + n);
}
```

- It is very simple what is that we are trying to do.
- To check whether the given number from keyboard is zero or not we need to execute that statement at least once without checking any condition.
- So we are not at all checking any condition we just executes the statement for once then we are checking the condition.
- So that part executes at least once.
- In this kind of scenario we should go for do-while loop.
- In the above code number of iterations are unknown.

- And we need to execute the statement at least once.
- User enters the number from keyboard at least once and condition is being checked.
- If non-zero value is entered then it again iterate the statements.
- If zero is entered then condition becomes false and it exits from the loop.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DoWhileEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int n;
            do{
                n=int.Parse(Console.ReadLine());
            }
            while(n!=0);

            Console.WriteLine("You Have Entered " + n);
        }
    }
}
```

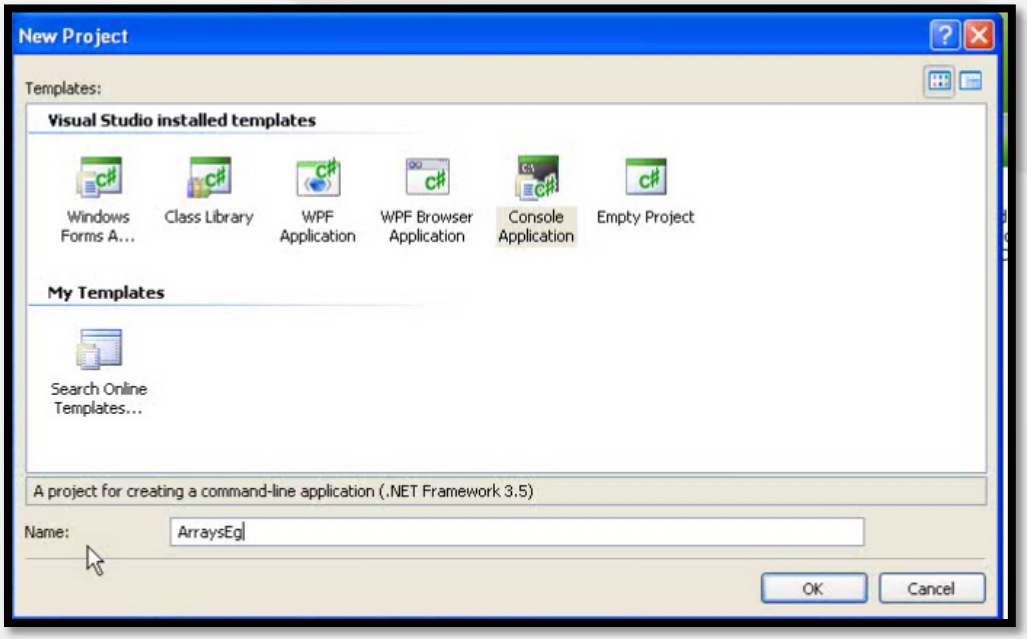
- Press F5.



- We gave 6, it checks 6 is not equal to zero.
- It asks us to give a number again, we gave 45.
- Again it asks us for give a number, we gave 23 and then 90.
- In this way goes on asking us to enter some number unless and until we give zero.
- If we give zero it says"You have Entered 0".
- This is how our do-while loop works.
- In this kind of situations we are going to use do-while loop.
- So do-while we are going to use whenever we have indefinite number of iterations and task must get executed at least once.
- So this is all about do-while loop.

CHAPTER 4: ARRAYS

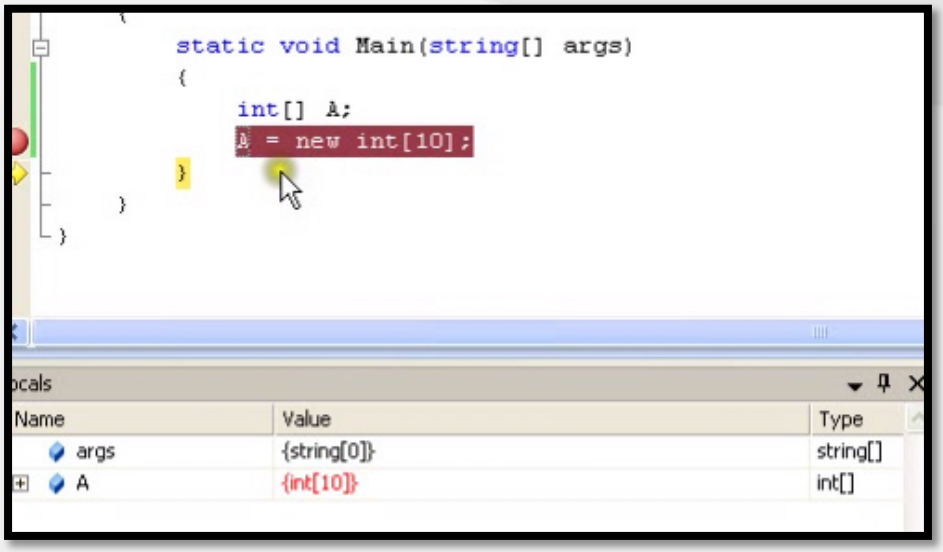
- In this chapter we are going to deal with arrays.
- What is an array?
- Array is nothing but collection of similar type of elements or collection of homogeneous data type elements.
- For example if we have a collection of elements.
- We want to store ids of 100 student, so it is very tough or difficult to us to declare 100 variables.
- So instead of that we'll go for the concept of Arrays.
- We'll try to understand the concept of arrays in which we are planning to store all integer type elements.
- Start new project.
- Program→ Visual C# Express Edition→File→New project→Console application.
- Name it as 'ArraysEg'.



- Array is nothing but similar type elements.
- So here we'll be declaring an array or we are trying to create an array.
- Say the name of the array is 'A'.
- And the size of the array that means we need to store 10 elements so let the size be 10.
- Syntax for declaring array is given below.

```
static void Main(string[] args)
{
    int[] A;
    A = new int[10];
}
```

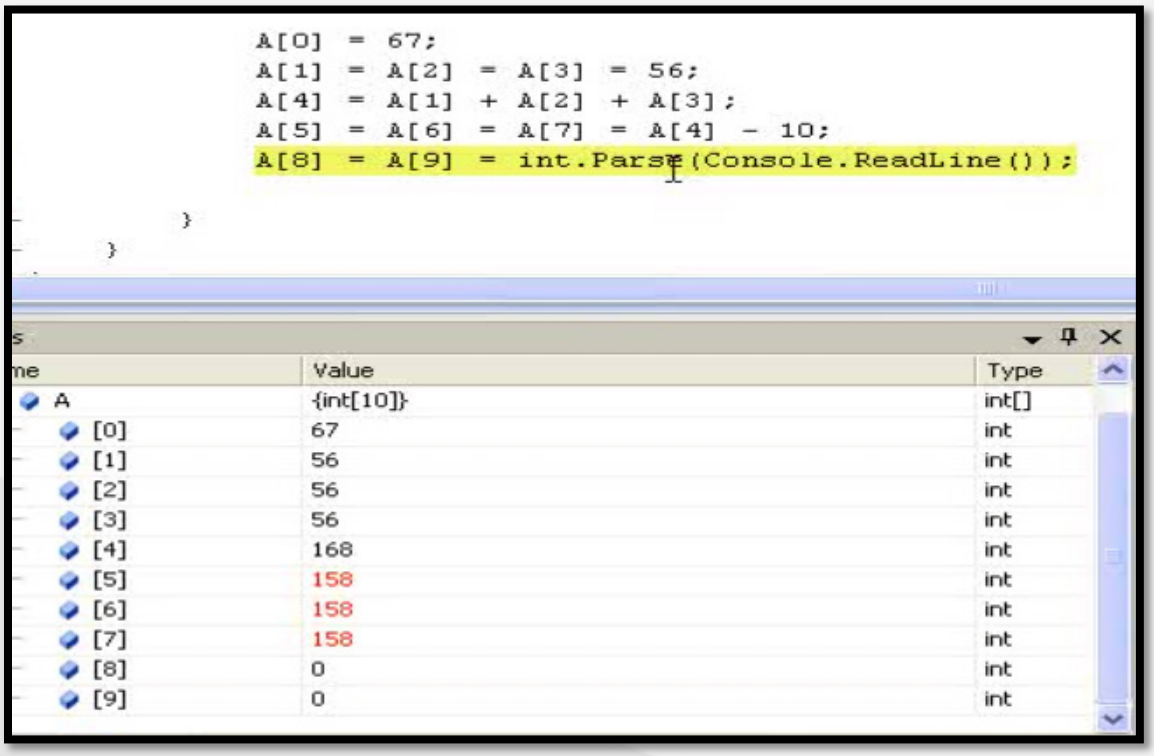
- Here in first statement defines that we are going to creating an array 'A' in which we are storing n number of integer values.
- In second we are defining the size of array for 10 integer variables.
- Now A is going to refer to the location where we have 10 memories allocated for integer variables.
- Now put break point and execute this and let us see how the memory allocation takes place.



- Press F11.
- Memory gets allocated see that 'int [10]'.
- If we observe the index we say that index value zero, at 0th location the default value is 'null'.
- So we have 10 locations index ranging from 0-9.
- If we say element at 0th location means we are referring to the first element.
- If we say element at 5th index means we are referring to the sixth element.
- This is how we've declared an array of 10 integers.
- To access the elements of an array we'll be using 'A[index]'.
- A [0] =67 that means we are trying to store 67 in 0th location.
- A[1]=A[2]=A[3]=56 this means we are trying to store 56 in 3rd location and that value trying to store in 2nd location and the same value trying to store at 1st index.
- That means our 2nd, 3rd, and 4th element will be 56.
- A[4]=A[1]+A[2]+A[3];
- That means the value of A [1], A [2], A [3] gets added and the result will be stored in A [4].
- That means at index 4 the location is 5th.

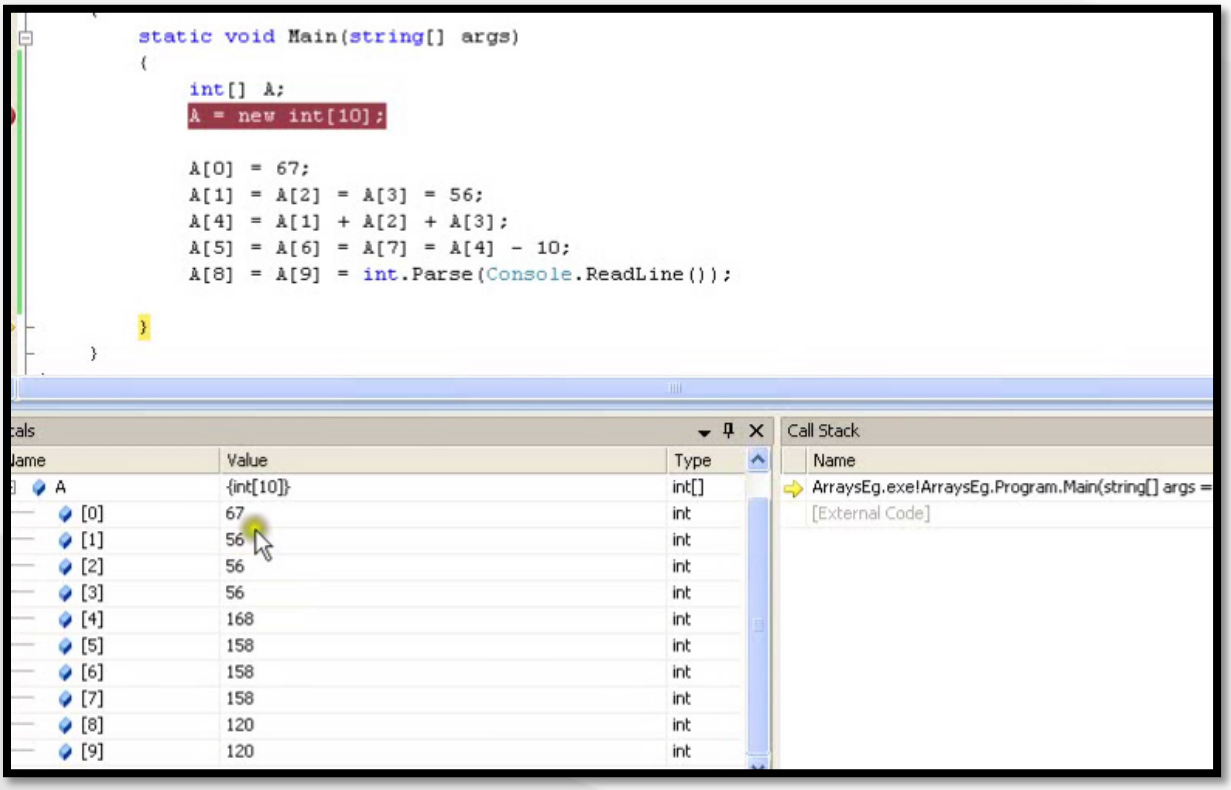
```
static void Main(string[] args)
{
    int[] A;
    A = new int[10];

    A[0] = 67;
    A[1] = A[2] = A[3] = 56;
    A[4] = A[1] + A[2] + A[3];
    A[5] = A[6] = A[7] = A[4] - 10;
    A[8] = A[9] = int.Parse(Console.ReadLine());

}
```

- But last 9th and 10th elements are accepting from keyboard.
- Put a break point and press F5.



- Now the value at the first location will be 67.
- In the second, third and fourth location will be 56.
- In fifth location i.e. A[4] will have the value 56+56+56 that will gives rise to 168.
- Now A[4] has 168.
- A[4]-10 means 168-10 i.e. 158 trying to store in A [5], A [6], A [7].
- Finally for last two locations we are asking the end user to give the value from keyboard.
- Press F11.

- Press enter.



- If we observe all the locations got filled with all the elements.
- This is how we can access the elements or we can store the values in an array.
- Now we want to display all the values.
- So we can simply write 'Cosole.WriteLine (A[0])' for first element.
- 'Cosole.WriteLine (A[1])' for second element.
- 'Cosole.WriteLine (A[2])' for third element.
- Like that we need to write till 9th location.
- If we observe everything is same only index value varying from 0 to 9.
- So we can re-write this and we can use the concept of 'for loop'.
- For display all the elements we have code below.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(A[i]);

}
Console.ReadLine();
```

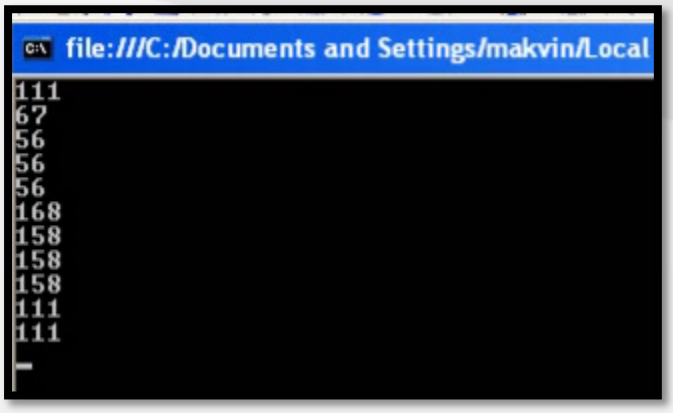
- Press F5.

- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace ArraysEg
{
    class Program
    {
        static void Main(string [] args)
        {
            int [] A;
            A = new int [10];
```

```
            A [0] = 67;
            A [1] = A [2] = A [3] = 56;
            A [4] = A [1] + A [2] + A [3];
            A [5] = A [6] = A [7] = A [4] - 10;
            A [8] = A [9] = int.Parse(Console.ReadLine());



            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(A [i]);

            }
            Console.ReadLine();

        }
    }
}
```

- Press F5.



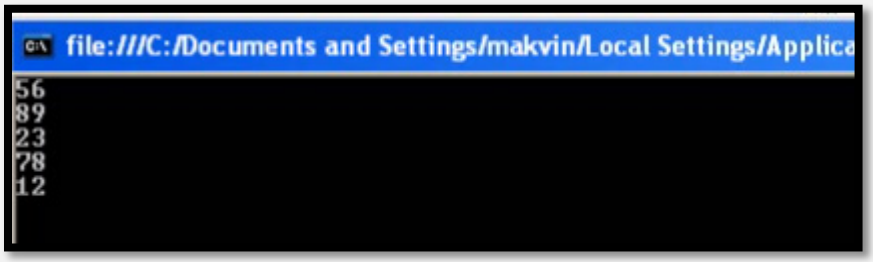
- This is how we can access the values.

CHAPTE 5 : UNDERSTANDING THE CONCEPT OF FOREACH LOOP AND SYSTEM.ARRAY METHODS IN C#.

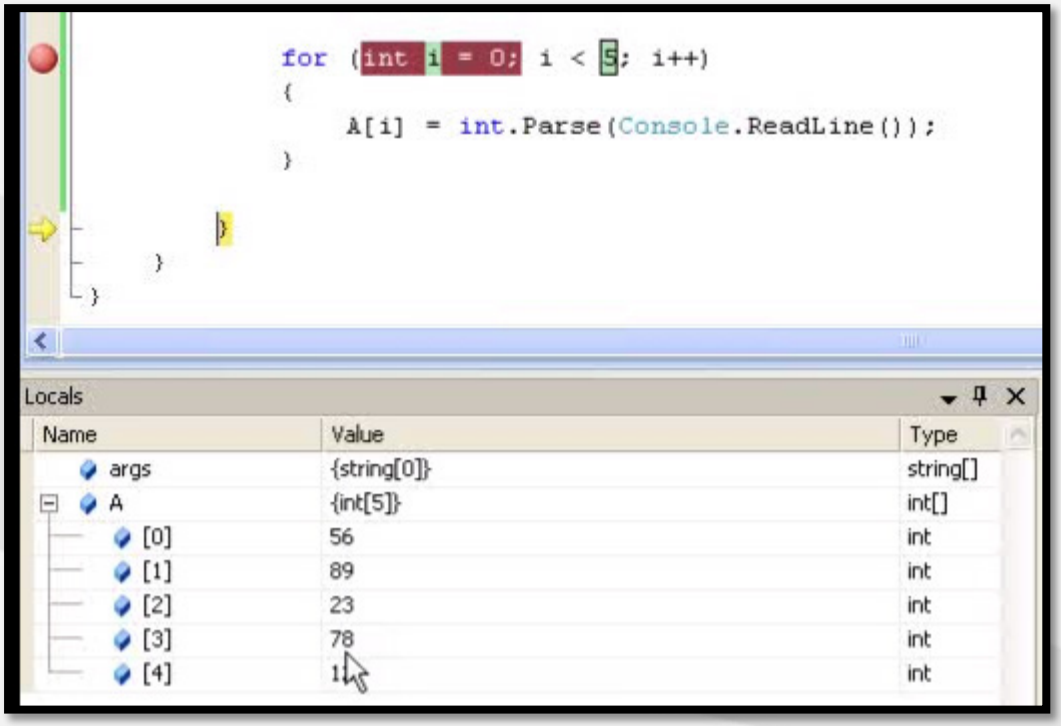
- Start new project.
- Program→ Visual C# Express Edition→File→New project→Console application.
- Give a name.
- In our earlier example we created an array and we stored all the values directly in our program.
- We've assigned values from our program we've hard code that.
- Now we'll write the program in which it should accept all the values from keyboard.
- Now let us declare an array of size 5.
- Int [] A=new Int [5];
- This is another way of declaring array.
- Now we need to accept all the values from the keyboard.
- Now we want to accept the values from the keyboard.
- So as to display all the values of an array we've used 'for loop'.
- In the same way if we want to accept all the values from the keyboard then we need to use 'for loop' again.

```
int[] A = new int[5];


for (int i = 0; i < 5; i++)
{
    A[i] = int.Parse(Console.ReadLine());
}
```

- Now put a break point and press F5.



- We have accepted all the values from the keyboard.



- If we observe one thing we need to remember the size of the array.
- If size of the array is 5 then we need to write 5 at condition i.e. i<5.
- Instead of remembering the size of array we can simply write **A. Length** at 5.
- So it will automatically take the length of an array.
- If it is 5 then it takes 5.
- If it is 6 then it takes 6.

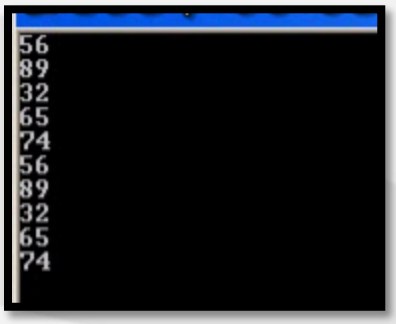
- If it is 100 then it takes 100 as size.
- Using this we are trying to accept the values from keyboard.

```
for (int i = 0; i < A.Length; i++)
{
    A[i] = int.Parse(Console.ReadLine());
}
```

- Now we can use same 'for loop' for displaying all the elements.

```
for (int i = 0; i < A.Length; i++)
{
    Console.WriteLine(A[i]);
}
```

- Press F5.



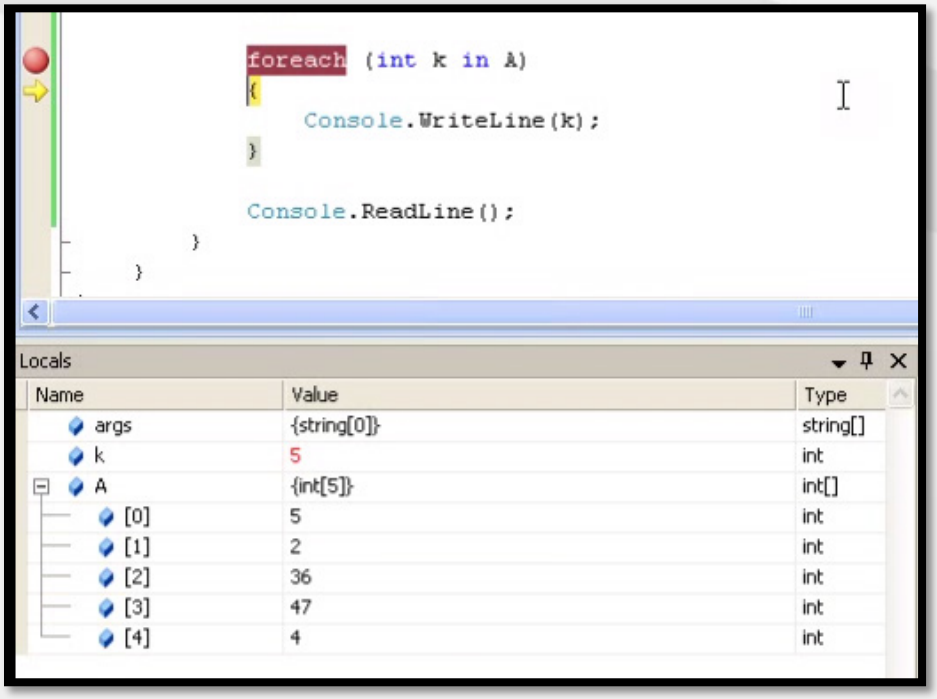
- This is how we displays all the elements that means we are iterating through all the elements one after the other from location 0 to till last location.
- To iterate through all the element of an array or to iterate through list of elements or list of objects we can use 'foreach loop'.

```
foreach (int k in A)
{
    Console.WriteLine(k);
}
```

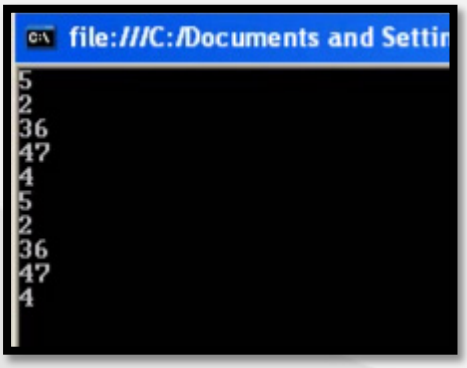
- It will first extract the top element of the array and it will store it in k.

- And we are trying to display that.
- Now it will go for next index i.e. index 1, it will take the element and put it in k.
- And we are displaying it.
- In this way it is going to extract the element one after the other from an array and going to store it in k.
- So in this way whatever the task we are performing using 'for loop' we can use it using 'foreach loop'.
- Put a break point and let us see how it works.
- Press F5.

- Press Enter.



- Now if we observe in ‘A’ we have 5 elements.
- Now it will extract 5 and it will try to stores in ‘k’.
- Initially k is 0, it extracts 5 and it will try to stores in ‘k’.
- Now ‘k’ value becomes 5.
- And it will displays 5.
- Next element is 2, it will extracts 2 and it will stores in k and displays it.
- In the same way it will put all remaining elements in ‘k’ and displays it one after the other.



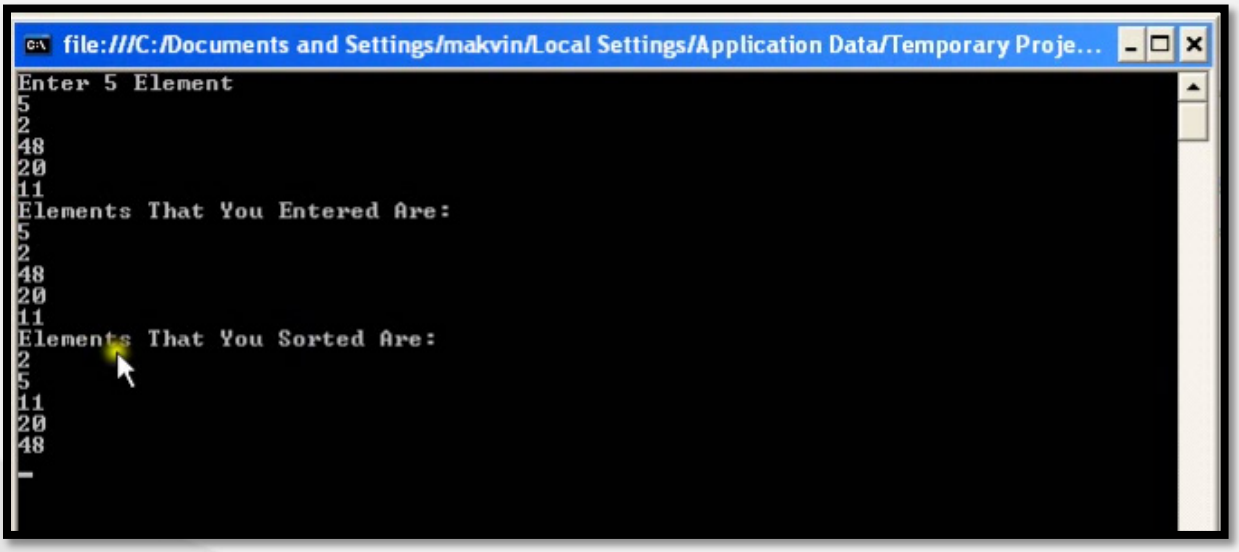
- This is how our foreach works.
- Foreach loop works for displaying the elements or to iterating through the set of elements.
- And we can use only for iterations not for storing.
- For example while iteration we can filter something.
- We want to filter out even elements.
- From the array we will display only even elements below.

```
foreach (int k in A)
{
    if(k%2==0)
    Console.WriteLine(k);
}
```

- So it is going to display only even elements.
- We'll be use foreach very frequently where we try to handle array of objects, list of employees, iterate through all the employees, list out the employees whose salary is more than 5000 and in many scenario we are going to use this foreach loop.
- So this how our foreach loop works.
- Now we want to sort this elements.
- We have many kind of sorting techniques to sort the elements.
- Like bubble sort, merge sort, selection sort.
- So we need to write big programs for that.
- Here we need not to write such kind of big programs.
- But we need to simply call a method to sort them.
- So we'll call a method which presents in array class **Array.sort(A)**.
- We are passing array name i.e. A.
- Now our elements got sorted.
- After sorting we want to display them.

```
Array.Sort(A);

Console.WriteLine("Elements That You Sorted Are:");
for (int i = 0; i < A.Length; i++)
{
    Console.WriteLine(A[i]);
}
```

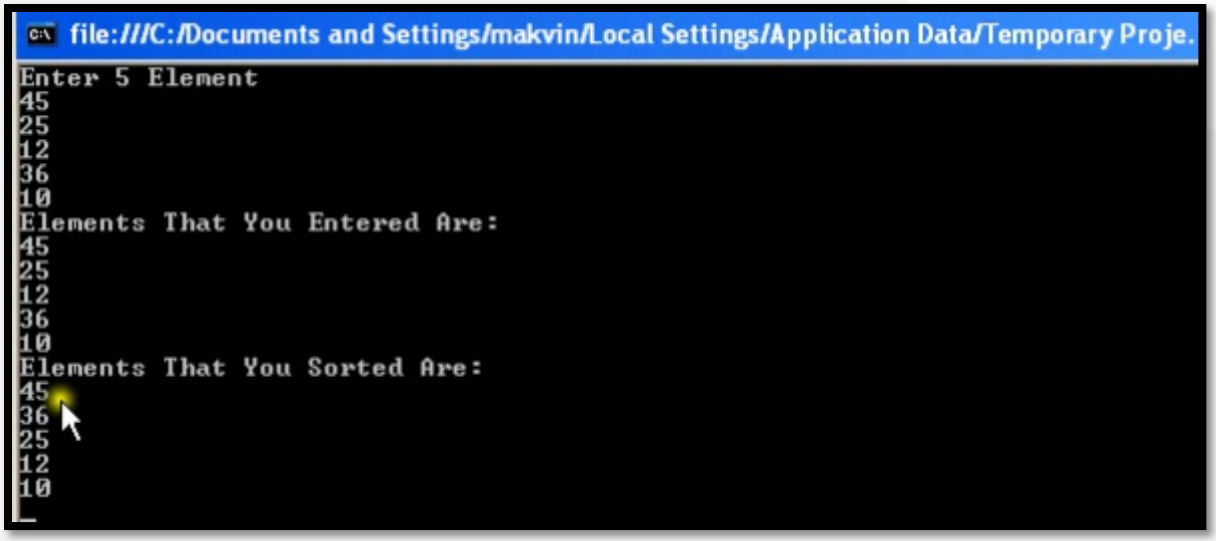
- We have given some messages to enter, display and for sorting.
- Press F5.



- We got the elements in ascending order.
- By default it sort the elements in ascending order.
- If we want to arrange them in descending order then we need to reverse them after sorting.
- See below.

```
Array.Sort(A);
Array.Reverse(A);

Console.WriteLine("Elements That You Sorted Are:");
for (int i = 0; i < A.Length; i++)
{
    Console.WriteLine(A[i]);
}
```

- Now press F5.



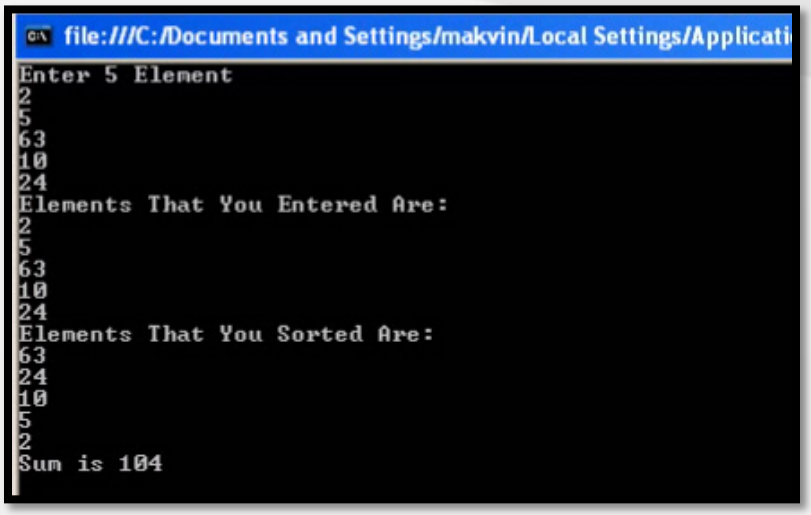
- Now we got the elements in descending order.
- This is our sorting technique using arrays.
- If we want sum of all the elements of array.
- It is very simple we need to use same for loop.
- Before that we need to declare an integer variable.
- Int sum=0;
- In loop instead of displaying all the values we need to write sum=sum+A[i].
- Means it's going to add all the values of the array.
- And we are trying to displays it.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ArraysEg2
{
```

```
class Program
{
    static void Main(string [] args)
    {
        //int [] A;
        //A = new int [5];

        int [] A = new int [5];

        Console.WriteLine("Enter 5 Element");
        for (int i = 0; i < A.Length; i++)
        {
            A [i] = int.Parse(Console.ReadLine());
        }

        Console.WriteLine("Elements That You Entered Are:");
        for (int i = 0; i < A.Length; i++)
        {
            Console.WriteLine(A [i]);
        }

        Array.Sort(A);
        Array.Reverse(A);


        Console.WriteLine("Elements That You Sorted Are:");
        for (int i = 0; i < A.Length; i++)
        {
            Console.WriteLine(A [i]);
```

```
        }
        int sum = 0;
        for (int i = 0; i < A.Length; i++)
        {
            sum = sum + A [i];
        }
        Console.WriteLine("Sum is " + sum);

        //foreach (int k in A)
        //{
        //    if(k%2==0)
        //    Console.WriteLine(k);
        //}

        Console.ReadLine();
    }
  }
}
```

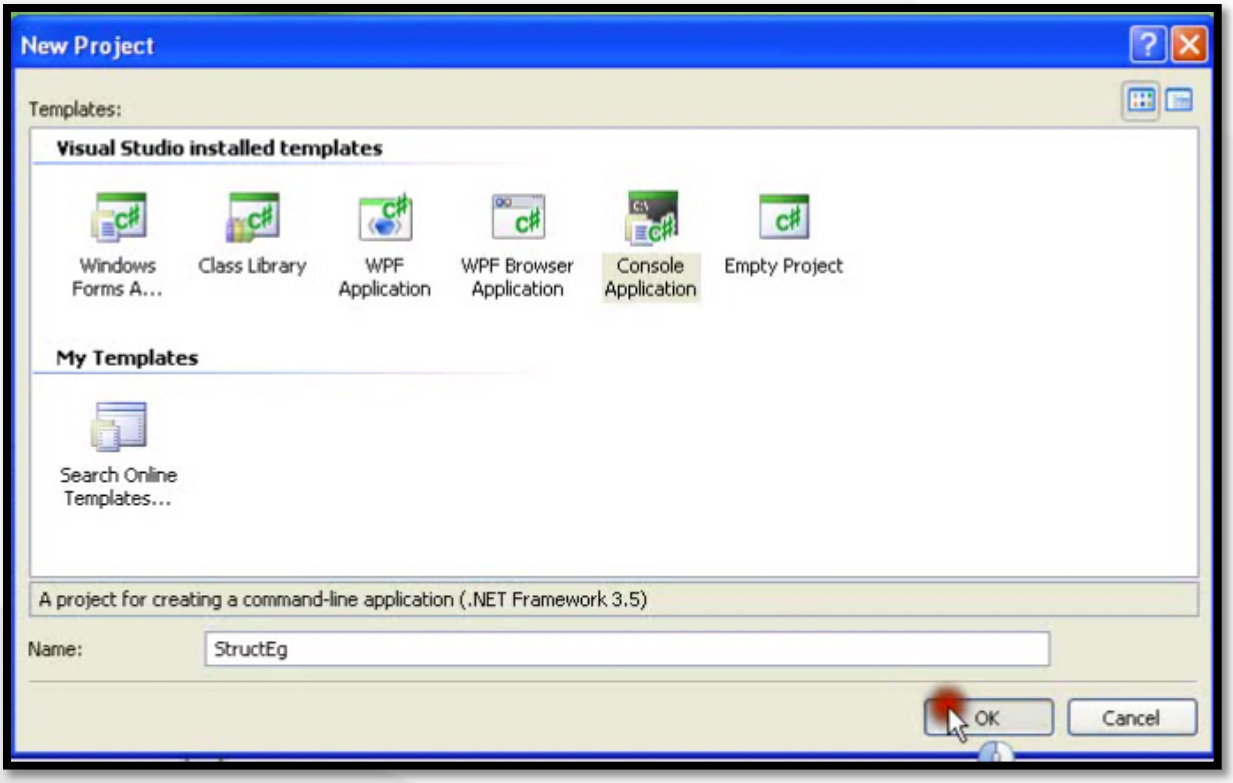
- Now press F5.



- We've entered the elements.
- Arranged in descending order.
- And sum is 104.
- This is very important why because when you developing some applications you might be coming across various points where we need to perform this kind of summation.
- Take an example of shopping card application where we have checked for various number of products and it is going to display all the products in a grid with the amount that we need to pay on right column.
- We need to do sum all the values of that columns and need to display the grand total at the footer.
- So same logic we need to use.
- So this logic will come across many situations once you going for real time development.

CHAPTER 6 : UNDERSTANDING THE CONCEPT OF STRUCTURES

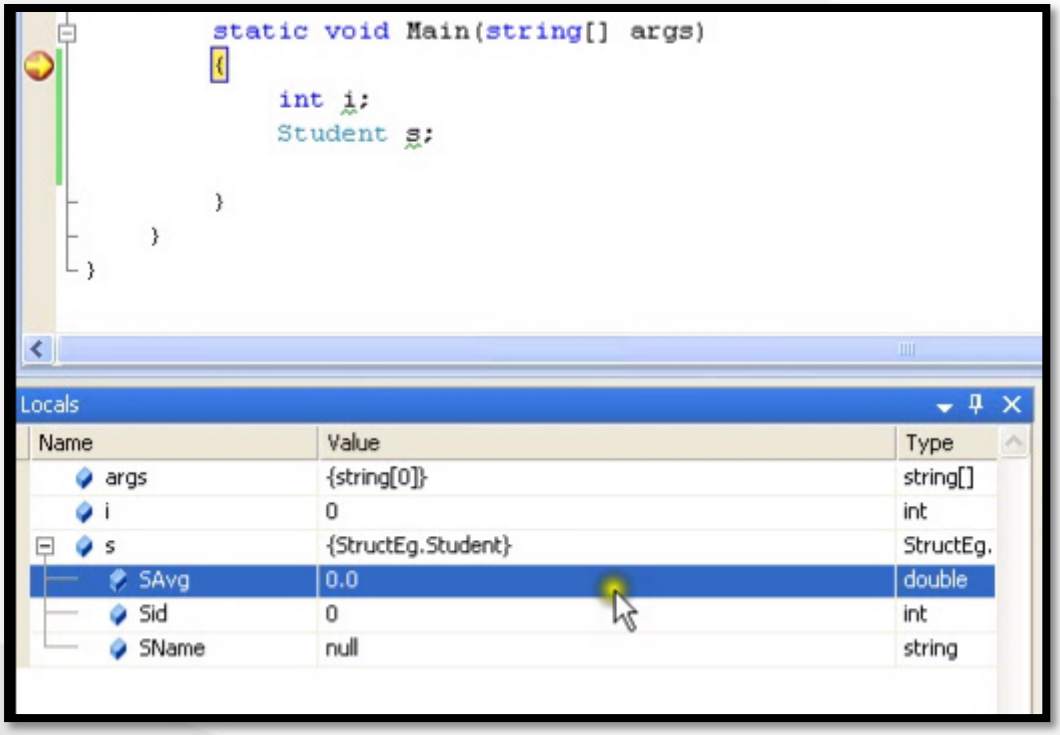
- In this chapter we'll going to deal with structures.
- We've seen arrays.
- Arrays are collection of homogenous data type elements.
- That means if we want to store information about student details we need to use 3 different types of array
- One for student id of integer type.
- Second is student name of string type.
- And third is student marks of double type.
- We cannot store these different types in a single array element why because array is collection of homogeneous data types.
- But we have a solution with the help of **Structures**.
- What is structure?
- Structures is collection of similar and dissimilar data type elements or heterogeneous data type elements.
- Start new project.
- Program→ Visual C# Express Edition→File→New project→Console application.
- Give a name.



- Press Ok.
- We can also say structure is a user defined data types.
- Like we have integer it is a predefined data type.
- We can create the variables of integer type.
- In the same way we can define the structure and we can create the variables of structures.
- We need to define structure of the student.
- Structure of the student includes student id, student name, and student marks.
- We'll be declaring structure above the class.
- Till now we've learnt written the code inside the class.
- Now we'll writing some code outside the class.
- Now we'll defining the structure.

```
namespace StructEg
{
    struct Student
    {
        public int Sid;
        public string SName;
        public double SAvg;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Student s;

        }
    }
}
```

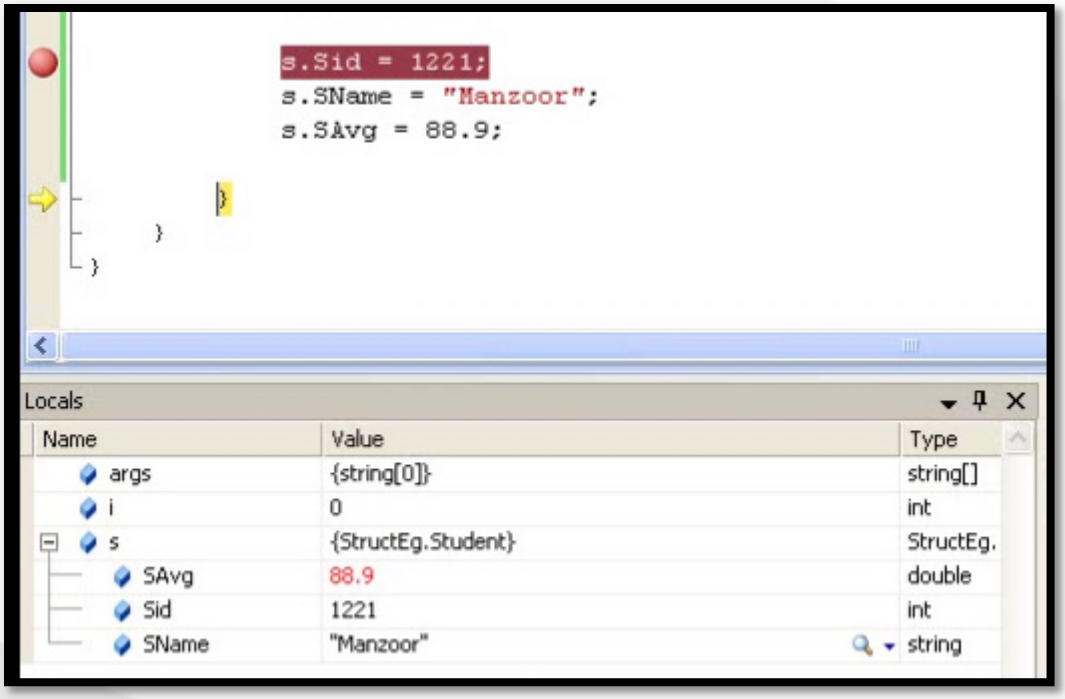
- This is the structure of student.
- This is the collection of similar and dissimilar data types.
- We've integer, string and double.
- Means heterogeneous data type elements.
- In class we declared the variable for structure student.
- Like if we want to declare the variable of integer then we write int i.
- In the same way we declared the variable of student.
- Put a break point and see the memory allocation for 'i and s'.



- See the difference for i it allocates single memory allocation of type integer.
- But for structure variable s it allocates 3 memory allocation with different data types.
- S is a structure variable and we have three fields in it Sid, SName, SAvg.
- We can access these Sid, SName, SAvg with the help of 'Period operator' or 'dot operator'.
- Like I was accessing the elements of an array using indexes A[0] where '0' is an index.
- In the same way we can access fields of structure using 'Period operator'.

```
s.Sid = 1221;
s.SName = "Manzoor";
s.SAvg = 88.9;
```

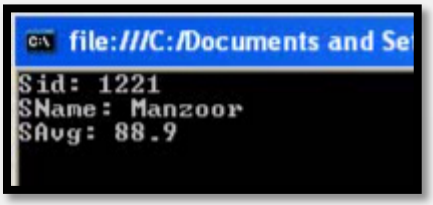
- Put a break point and see the execution.



- See that how all values got stored in different fields.
- Now we'll display them using simple Cosole.WriteLine () method.

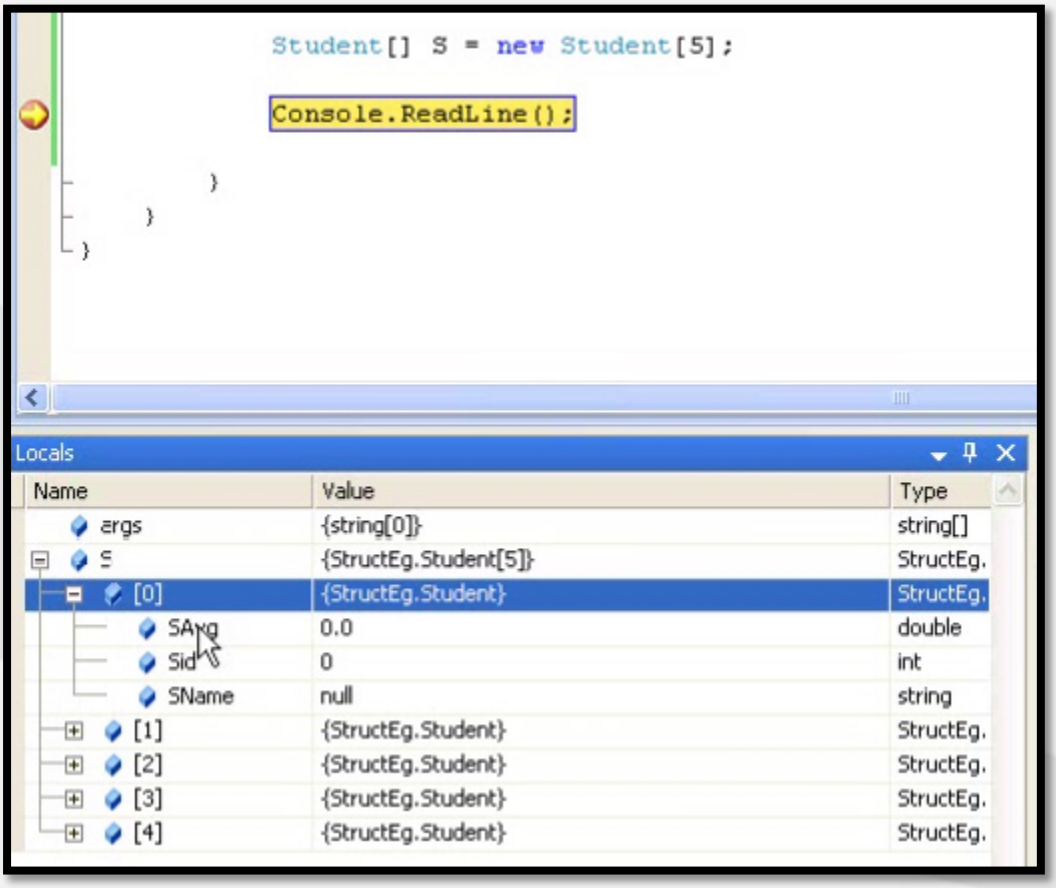
```
Console.WriteLine("Sid: " + s.Sid);
Console.WriteLine("SName: " + s.SName);
Console.WriteLine("SAvg: " + s.SAvg);
```

- Press F5.



- This is how we are creating single student.
- If we have two or three student we need to create two or three variables.
- If we have 100 students we need not to create so many variables.

- Here we can apply the concept of **Arrays to the Structures.**
- It is simple now we'll declare an array of student in the same way used to declare an array of integer.
- We are trying to store the records of 5 students.
- Let us see how does the memory allocation takes place.



- Structure variable has 5 indexes S[0]-S[4] and each index has three members.
- We are accessing the elements with the help of index.
- Now each element is of type structure.
- So S[0].Sid, S[0].SName in this way we can access.
- Now we'll try to store records of two students below.

```
Student[] S = new Student[5];

S[0].Sid = 1221;
S[0].SName = "Manzoor";
S[0].SAvg = 88.5;


S[1].Sid = 1221;
S[1].SName = "Manzoor";
S[1].SAvg = 88.5;
```

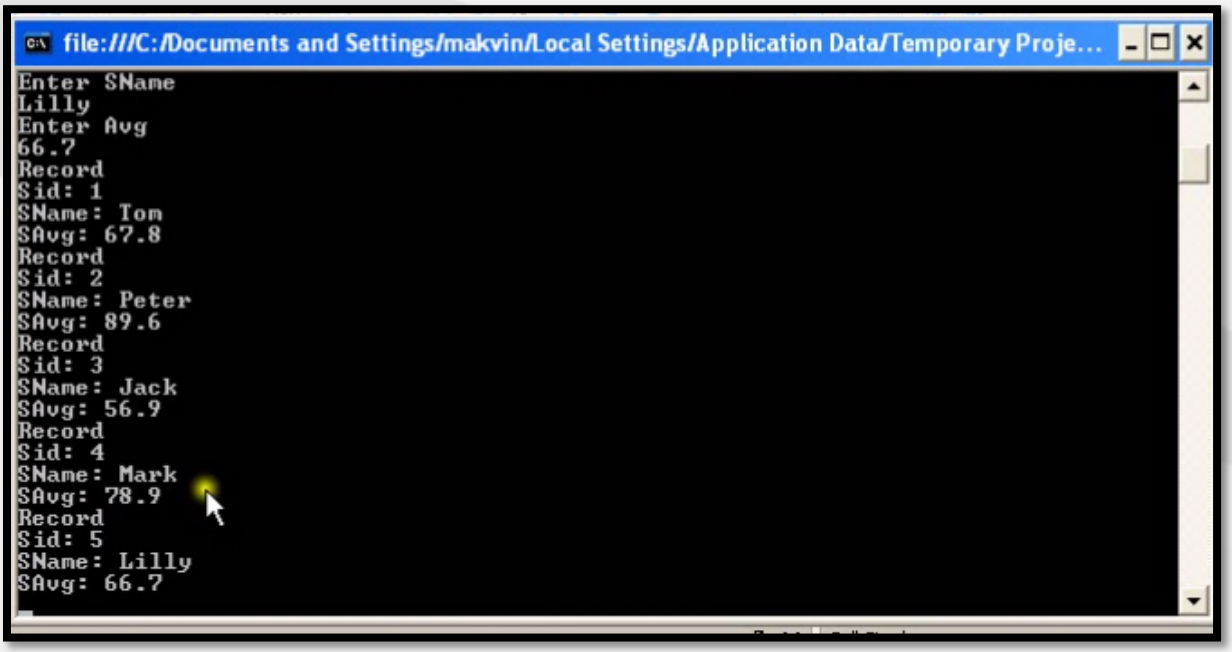
- Instead of writing same statement again and again we can give index value in for loop and accessing the values from keyboard.
- We'll try to accept the values or information about 5 students.
- And we are trying to store in an array of structure s.
- So we can proceed with 'for loop'.

```
Student[] S = new Student[5];

for (int i = 0; i < S.Length; i++)
{
    Console.WriteLine("Enter Sid");
    S[i].Sid = int.Parse(Console.ReadLine());

    Console.WriteLine("Enter SName");
    S[i].SName = Console.ReadLine();

    Console.WriteLine("Enter Avg");
    S[i].SAvg = double.Parse(Console.ReadLine());
}
```

- Now we want to display those things.
- Same procedure we can go for this using for loop.

```
for (int i = 0; i < S.Length; i++)
{
    Console.WriteLine("Record");
    Console.WriteLine("Sid: " + S[i].Sid);
    Console.WriteLine("SName: " + S[i].SName);
    Console.WriteLine("SAvg: " + S[i].SAvg);
}
```

- That means in first iteration get the record of 0th student.
- In second iteration we get the record of 1st student.
- Then second then third then fourth till four it is going to work.
- Now let us execute this.
- Press F5.
- We need to input 5 records.

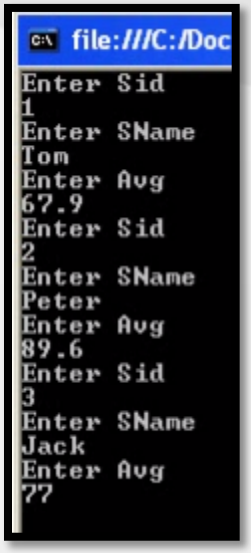


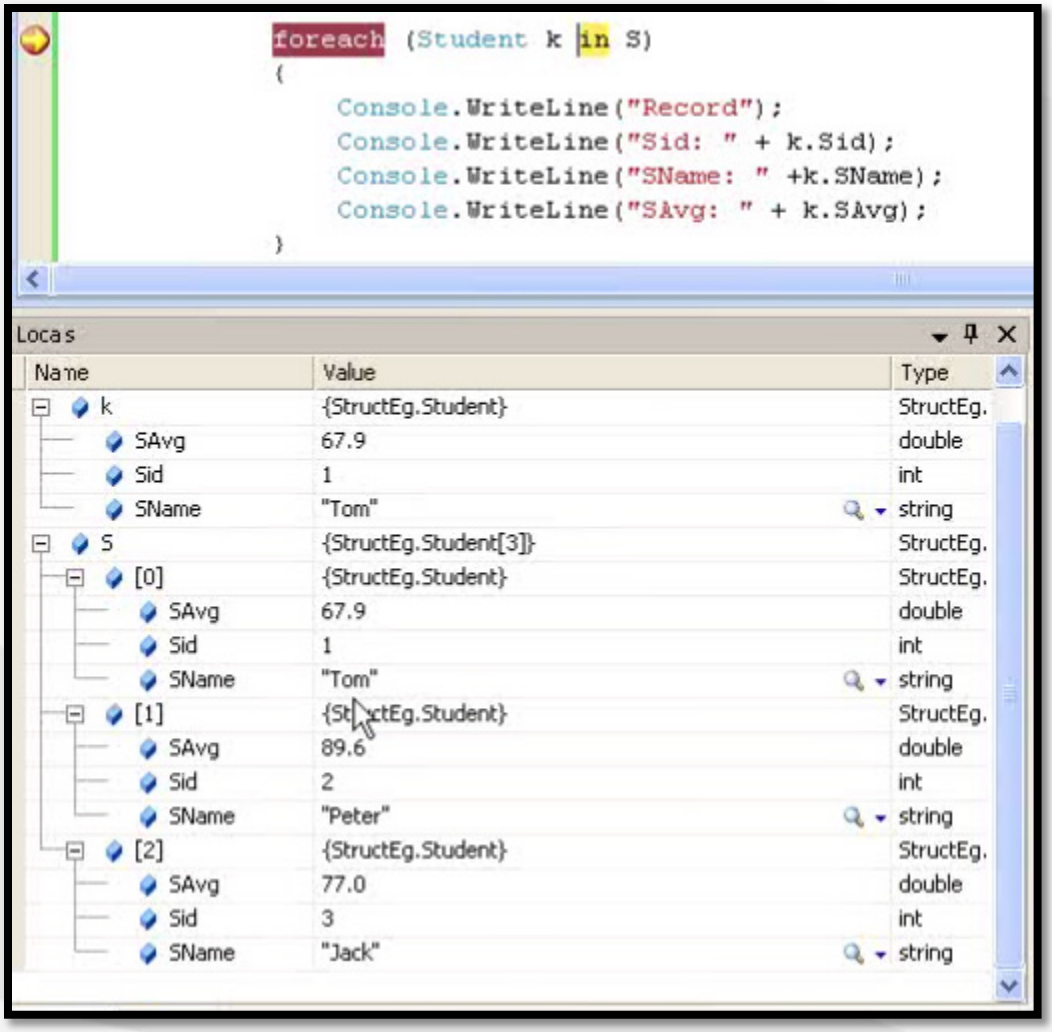
- Here we displays all the records.
- This how we can work with arrays of structure.
- That means we've array of student.

- Same thing we can work with foreach loop.
- In foreach here the type is not int.
- In our earlier program or arrays we are working with integer types.
- But here the type is ‘Student’.
- And the collection is ‘S’.

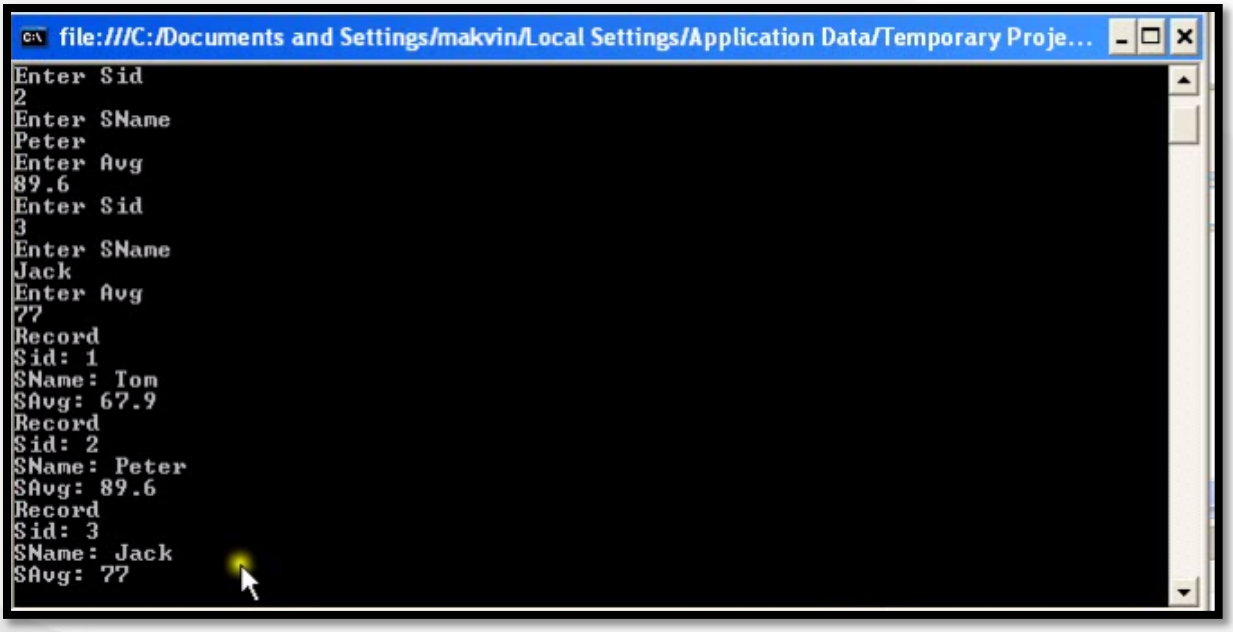
```
foreach (Student k in S)
{
    Console.WriteLine("Record");
    Console.WriteLine("Sid: " + k.Sid);
    Console.WriteLine("SName: " +k.SName);
    Console.WriteLine("SAvg: " + k.SAvg);
}
```

- In this scenario foreach works as, it takes the each record from collection S and stores in k and displays it.
- Put a break point and see the execution.
- Press F5.
- It asks for enter 3 student records.





- If we observe we have records of 3 students.
- Press F11.
- Even we have one more record ‘k’ which is null initially.
- It will picks the first record and copy it to ‘k’ and it will display ‘k’.
- We can see in above snap.
- In the same it pics all students records from collection and stores in ‘k’ then displays it one after the other.



- Here we can see all records of students.
-
- Now we'll display the student records who got average more than 80.
- Simply we need to filter.

```
foreach (Student k in S)
{
    if (k.SAvg >= 80)
    {
        Console.WriteLine("Record");
        Console.WriteLine("Sid: " + k.Sid);
        Console.WriteLine("SName: " + k.SName);
        Console.WriteLine("SAvg: " + k.SAvg);
    }
}
```

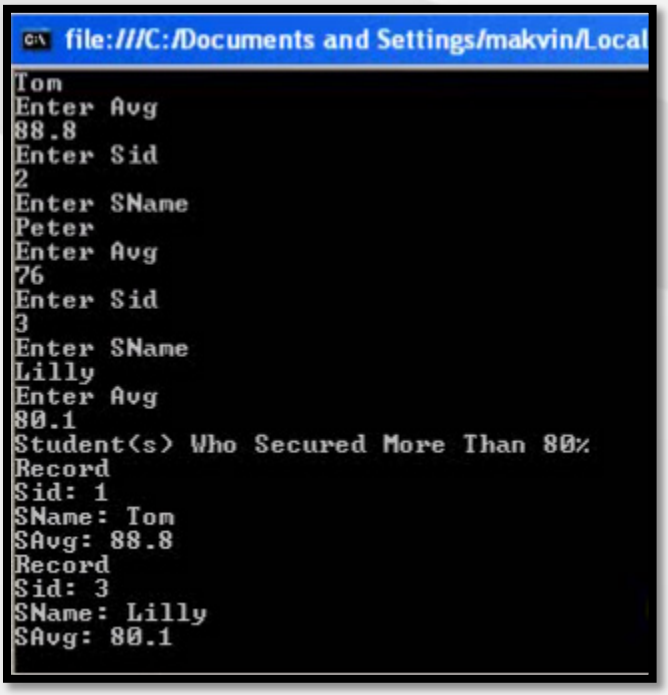
- It filters the records with student average.
- And complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StructEg
{
    struct Student
    {
        public int Sid;
        public string SName;
        public double SAvg;
    }
    class Program
    {
        static void Main(string [] args)
        {

            //Student s;

            //s.Sid = 1221;
            //s.SName = "Manzoor";
            //s.SAvg = 88.9;


            //Console.WriteLine("Sid: " + s.Sid);
            //Console.WriteLine("SName: " + s.SName);
            //Console.WriteLine("SAvg: " + s.SAvg);
```

```
Student [] S = new Student [3];

for (int i = 0; i < S.Length; i++)
{
    Console.WriteLine("Enter Sid");
    S [i].Sid = int.Parse(Console.ReadLine());

    Console.WriteLine("Enter SName");
    S [i].SName = Console.ReadLine();

    Console.WriteLine("Enter Avg");
    S [i].SAvg = double.Parse(Console.ReadLine());
}

Console.WriteLine("Student(s) Who Secured More Than 80%");
foreach (Student k in S)
{
    if (k.SAvg >= 80)
    {
        Console.WriteLine("Record");
        Console.WriteLine("Sid: " + k.Sid);
        Console.WriteLine("SName: " + k.SName);
        Console.WriteLine("SAvg: " + k.SAvg);
    }
}
```

```
            //for (int i = 0; i < S.Length; i++)
            //{
            //    Console.WriteLine("Record");
            //   Console.WriteLine("Sid: " + S [i].Sid);
            //   Console.WriteLine("SName: " + S [i].SName);
            //   Console.WriteLine("SAvg: " + S [i].SAvg);
            //}

            Console.ReadLine();

        }
    }
}
```

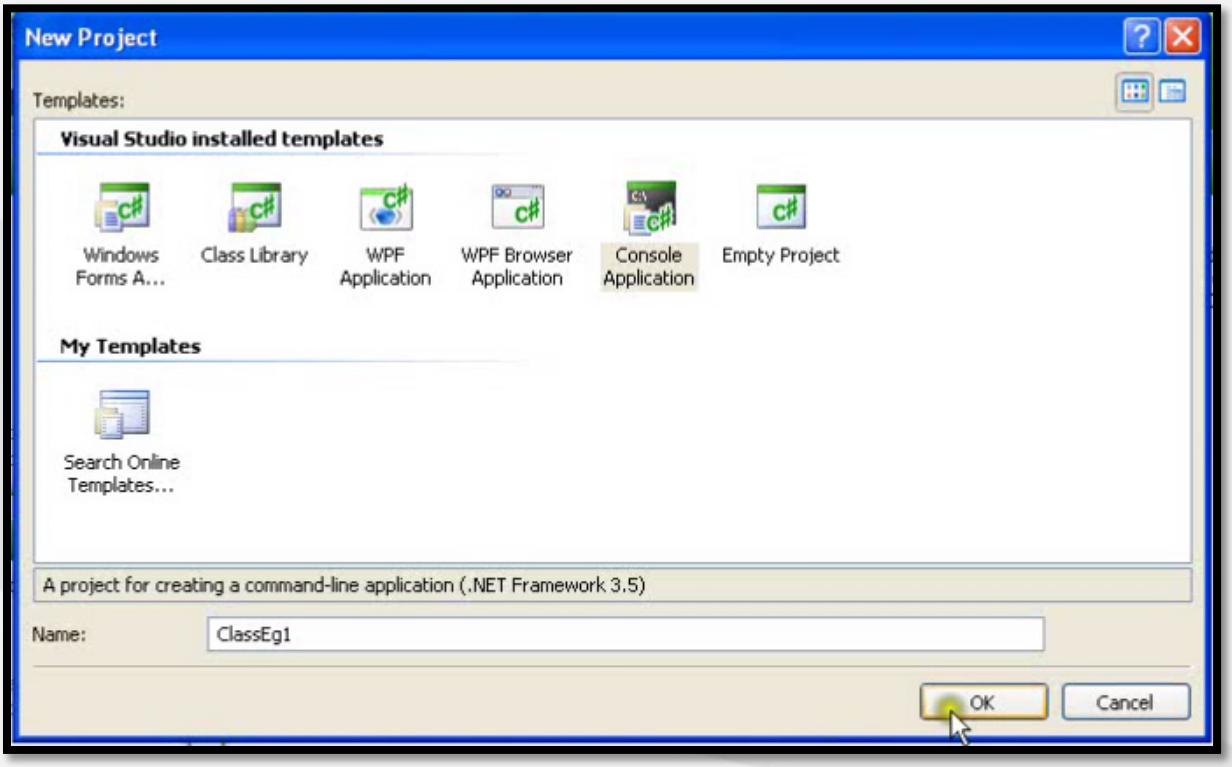
- Now Press F5.



- See the details we've entered 3 records but as per average it displays 2 records.
- Record 1 and 3 who secured more than 80%.
- In this chapter we have seen structures and various methods to access it.

CHAPTER 7 : CLASS

- **Class, Object, Public, Private, Method, Constructor, and constructor overloading in C#.**
- In this chapter we'll see what a class is, public and private members, and methods.



- Select new project.
- File→new project→Console application.
- We need to define a class.
- **CLASS** is nothing but it is a blue print of an object.
- As we have define the structure in the same way we define the class.

```
class Customer
{
    int AcNo;
    string Name;
    double Bal;
}
```

- This is the customer of a bank.
- Customer of bank may have various fields.
- So for our demonstration we are selecting three fields.
- One is AcNo, second is Name, and one more is Balance.
- As per the rules we should have all the fields of our class is private.
- To make all the fields of our class private we need to declare them with the help of private keyword.
- Or by default members of the class are private.

```
class Customer
{
   private int AcNo;
    string Name;
    double Bal;
}
```

- If we do not write any things or any access specifiers (i.e. public, private) by default the fields of the class are private.
- **Private** means cannot be accessible from outside the class.
- And **Public** means can access it from outside of the class.
- Now in this class we need to declare few methods.
- Methods are nothing but behaviours.
- So whenever a new customer comes to the bank or new customer wants to create an account so we need to assign account number, name and balance to that particular customer.

- So whenever a new customer comes to the bank to create an account we need to initialize their account number, name and balance.
- For that let us write a behaviour.
- Behaviour is nothing but a method.
- As per the rules behaviours must be public.
- Whenever we create the customer we need to pass three parameters for account number, name and balance.

```
public void CreateCustomer(int a, string n, double b)
{
    AcNo = a;
    Name = n;
    Bal = b;
}
```

- If we observe we write 'void' because our method is not returning any value.
- Whereas it is taking three values, we call it as parameters.
- So it is taking three parameters one is for account number, another is name and one more is balance.
- So this is the method which we used to create an object.
- If we want to display the details of that particular customer or we can say balance enquiry.

```
public void BalEnq()
{
    Console.WriteLine("AcNO:{0} Name:{1} Bal:{2}",AcNo,Name,Bal);
}
```

- In this we neither passing any parameter nor returning any value.
- Just we displays account number, name and balance.
- This is our simple class with creation of customer and displaying the balance enquiry.

- Here we are not at all performing any kind of operations.
- Like amount withdraw or deposit etc.
- So this is the blueprint of customer.
- Any customer of the bank whether it is tom, jack, peter or any one any customer will have all this things.
- Now we need to create the object.
- In structure we calling creating the variable of structure.
- Here in classes we call it as creating the object of particular class.

```
class Program
{
    static void Main(string[] args)
    {
        Customer c;
        c = new Customer();
    }
}
```

- To create an object we are using the syntax similar to arrays.
- We can say that 'c' is the object of class customer.
- This is the way we can create the object.
- We have alternate way.

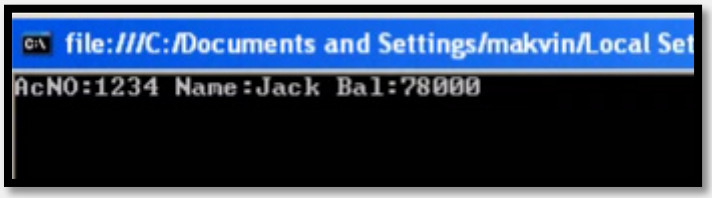
```
static void Main(string[] args)
{
    Customer c = new Customer();
}
```

- This is the easiest way of creating an object.
- Once the object is created if we want to access the member of class we can use period or dot operator.
- Like c.BalEnq().

- If we observe we will not show fields in their members because they are private by default.
- If we make them as public then we can access it.
- But as per the rules of a class objects should not have access the fields directly.
- That's way we making the fields as private.
- So as we might have observed in structures we declared all the fields of structure as public so that we should access them.

```
class Program
{
    static void Main(string[] args)
    {
        Customer c = new Customer();
        c.CreateCustomer(1234, "Jack", 78000);

        c.BalEnq();

        Console.ReadLine();
    }
```

- So this is our class and we creates the customer and we pass three parameters.
- With this we create the customer.
- To display the details we called BalEnq() method.
- Press F5.



- It displays all the details.
- This is simple class.

- If we observe very carefully this program has two ambiguities.
- We are making fields as private so that our object should not access them directly.
- Means they should not change the fields once the object gets created can we change the account number.
- For example if we go to bank we've created our account can we update the account number?
- End user will not have direct access to update the account number.
- One customer cannot have two account numbers of same account.
- Main intention is keeping all fields as private.
- So there is two ambiguities.
- One is we can re initializing the object by calling that method again and again.
- And another is we need to call the method create customer explicitly to initialize the object.
- Solution to this problem is **Constructor.**
- **Constructor** is a special method, same name as class name, may or may not have parameters, with no return type, and it is public in most of the cases.
- **Two functional features** of constructor are it invokes automatically whenever we create an object, and we cannot invoke explicitly.
- **Use of Constructor** is it used to initialize the object.
- In our earlier program we were facing two ambiguities one is we need to call create customer explicitly whenever we create the object.
- So that problem can be solve with this feature of constructor.
- It invokes automatically whenever we create an object.

- The next point is we can re initializing the object by calling that method again and again that problem is solved with this feature of constructor that we cannot invoke the constructor explicitly.
- Now we are declaring two constructors in class customer one with no parameter called as **Default constructor**.
- And another is with 3 parameters.

```
public Customer()
{
    Console.WriteLine("Hello World");
}

public Customer(int a, string n, double b)
{
    AcNo = a;
    Name = n;
    Bal = b;
}
```

- So whenever we have two constructors with different parameter, this concept we call it as **constructor overloading**.
- Now the question is whenever we create the object which constructor is invoked automatically?
- Definitely default constructor.
- Why because we are not passing any parameters while creating the object.
- Now we'll use second constructor.

```
static void Main(string[] args)
{
    Customer c = new Customer(1234, "Jack", 78000);
```

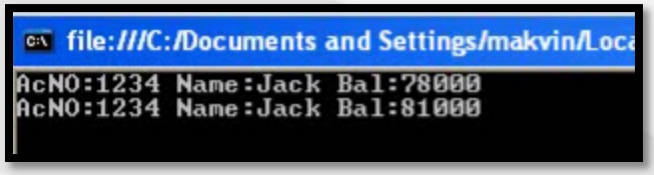
- Now this is going to automatically invoke parameterized constructor.
- Let us write one more method deposit.

```
public void Deposit(double amt)
{
    Bal = Bal + amt;
}
```

- Whenever we call this deposit method our balance get incremented
- Our new balance will be old balance added with this amount.
- Now will deposit 3000 rupees.
- And we'll call balance enquiry.

```
c.BalEnq();

c.Deposit(3000);

c.BalEnq();
```

- Press F5.



- 3000 got deposited.
- Now our new balance is 81000.
- In the same way we can write withdraw method.

```
public void WithDraw(double amt)
{
    Bal = Bal - amt;
}
```

- Now we'll withdraw 5000 rupees.
- And call the balance enquiry method.

```
c.WithDraw(5000);

c.BalEnq();
```

```
file:///C:/Documents and Settings/makvin/Local Set
AcNO:1234 Name:Jack Bal:78000
AcNO:1234 Name:Jack Bal:81000
AcNO:1234 Name:Jack Bal:76000
```

- After withdraw 5000 our new amount is 76000.
- This is what we've learnt about constructor.

CHAPTER 8 : METHOD OVERLOADING OR STATIC POLYMORPHISM

- The process of choosing particular method among various methods with same name but with different parameters at compile time is known as **Method overloading or static polymorphism**
- Static Polymorphism stands for static means at compile time polymorphism means many forms.
- With the help of previous example we'll see the complete meaning of Method overloading or Static Polymorphism.
- In that program we've separate methods for deposit and balance enquiry as well as withdraw and balance enquiry.
- So whenever we want to deposit we need to call deposit method then balance enquiry.
- And for withdraw we need to call withdraw method then balance enquiry separately.
- Now we'll try to define new balance enquiry method with two parameters.
- One parameter is amount and another is flag.
- We'll just pass a parameter amount and another parameter is flag 'D' for deposit and 'W' for withdraw.

```
public void BalEnq()
{
    Console.WriteLine("AcNO:{0} Name:{1} Bal:{2}",AcNo,Name,Bal);
}

public void BalEnq(double amt, string f)
{
    if (f == "D")
    {
        //Bal = Bal + amt;
        Bal += amt;
    }
    else if (f == "W")
    {
        Bal -= amt;
    }
    else
    {
        Console.WriteLine("INVALID Flag");
    }

    Console.WriteLine("AcNO:{0} Name:{1} Bal:{2}", AcNo, Name, Bal);
}
```

- Here we've two balance enquiry methods.
- One balance enquiry method using which we don't want to perform any operation just we display the information about balance.
- Another balance enquiry where we can perform deposit or withdraw.
- The second parameter of balance enquiry method is to refer either deposit or withdraw operation.
- If 'f' equals to 'D' then it will perform deposit operation.
- If 'f' equals to 'W' then it will perform withdraw operation.
- Otherwise it shows 'Invalid Flag'
- Now let we call new balance enquiry method.

```
c.BalEnq();

c.BalEnq(4500, "D");
```

- Execute this.

```
AcNO:1234 Name:Jack Bal:78000
AcNO:1234 Name:Jack Bal:81000
AcNO:1234 Name:Jack Bal:76000
AcNO:1234 Name:Jack Bal:80500
```

- Here we've deposited 4500 rupees.
- Put a break point and start execution.
- Now press F5.

```
public void BalEnq(double amt, string f)
{
    if (f == "D")
    {
        //Bal = Bal + amt;
        Bal += amt;
    }
    else if (f == "W")
    {
        Bal -= amt;
    }
    else
    {
        Console.WriteLine("INVALID Flag");
    }

    Console.WriteLine("AcNO:{0} Name:{1} Bal:{2}", AcNo, Name, Bal);
```

- It calls new balance enquiry method with two parameter.
- If it is 'D' it goes for incrementing the balance comes out of the block and displays the balance account number and name.

```
AcNO:1234 Name:Jack Bal:78000
AcNO:1234 Name:Jack Bal:81000
AcNO:1234 Name:Jack Bal:76000
AcNO:1234 Name:Jack Bal:80500
```

- So we've taken this example to explain that we can have more than one method with same name in a class but different parameters as we have more than one constructor with different parameters.
- This concept is called as 'method overloading'.
- Method overloading is also called as 'static polymorphism'.

- Static means ‘compile time’ poly means ‘many’ morphism means ‘forms’
- So in our program we have many forms.
- One balance enquiry with no parameters and another with two parameters.
- So we can understand that it is polymorphism.
- We call it as polymorphism because without executing the program at compile time itself we knew that balance enquiry with two parameters going to invoke balance enquiry method with two parameters.
- That means code for this method is generated at compile time itself that means it need not to wait for runtime.
- Our code gets generated at compile time itself.
- So at compile time we knew that what method is going to be invoked so we call it as static.
- Polymorphism means there are many forms of single balance enquiry.
- So we call it as ‘static polymorphism’.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassEg1
{
    class Customer
    {
        int AcNo;
        string Name;
        double Bal;
```

```
public Customer()
{
    Console.WriteLine("Hello World");
}

public Customer(int a, string n, double b)
{
    AcNo = a;
    Name = n;
    Bal = b;
}

public void Deposit(double amt)
{
    Bal = Bal + amt;
}

public void WithDraw(double amt)
{
    Bal = Bal - amt;
}

//public void CreateCustomer(int a, string n, double b)
//{
//    AcNo = a;
//    Name = n;
//    Bal = b;
//}
```

```
    public void BalEnq()
    {
        Console.WriteLine("AcNO:{0} Name:{1} Bal:{2}",AcNo,Name,Bal);
    }

    public void BalEnq(double amt, string f)
    {
        if (f == "D")
        {
            //Bal = Bal + amt;
            Bal += amt;
        }
        else if (f == "W")
        {
            Bal -= amt;
        }
        else
        {
            Console.WriteLine("INVALID Flag");
        }

        Console.WriteLine("AcNO:{0} Name:{1} Bal:{2}", AcNo, Name, Bal);
    }

}
class Program
{
    static void Main(string [] args)
```

```
        {
            Customer c = new Customer(1234, "Jack", 78000);

            //c.CreateCustomer(1234, "Jack", 78000);

            //c.BalEnq();

            //c.CreateCustomer(1234, "Jack", 80000);

            c.BalEnq();

            c.Deposit(3000);

            c.BalEnq();

            c.WithDraw(5000);

            c.BalEnq();

            c.BalEnq(4500, "D");

            Console.ReadLine();
        }
    }
}
```

CHAPTER 9 : THIS KEYWORD IN C#:

- In this chapter we'll see about 'this' keyword.
- Before we proceed for this keyword let us see what is the use of this keyword or where should we use this keyword, what is the necessity of this keyword.
- We'll just use previous program 'class customer' example.
- If we go for real time projects like if we are developing an application for bank.
- So bank would be giving a big form for customer where he need to fill 30-40 fields.
- So the problem is while initializing those fields we need to create few fields and we need to assign the values of variables to the fields.
- So we need to remember that 'a' must be assign to account number, 'n' must be assign to name and 'b' must be assign to balance.
- For example we have one more field 'contact' and we'll pass parameter for that contact.
- In a hurry by mistake if we assign 'n' to the contact field it could be a problem that may occur but our program is not going to stop us to doing so.
- But in place of contact it's going to store name.

```
class Customer
{
    int AcNo;
    string Name;
    double Bal;
    string Contact;

    public Customer()
    {
        Console.WriteLine("Hello World");
    }

    public Customer(int a, string n, double b,string c)
    {
        AcNo = a;
        Name = n;
        Bal = b;
        Contact = n;
    }
```

- So how do we avoid this kind of ambiguities?
- It's very simple instead of having various different names a, b and c lets have same name.
- For account number 'AcNo', for name 'Name', for balance 'Bal' and for contact 'Contact'.
- Same thing we need to replace inside but at left side variables we need to use 'this' keyword.
- Because this keyword refers to the object of current class, class in which we are using it.
- To avoid the ambiguities we use this keyword for field of class.

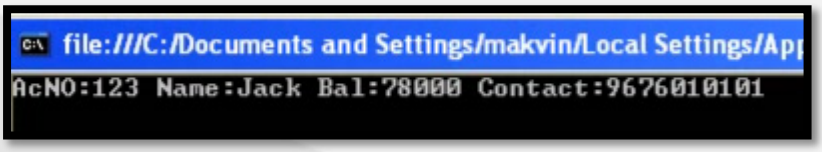
```
public Customer(int AcNo, string Name, double Bal,string Contact)
{
    this.AcNo = AcNo;
    this.Name = Name;
    this.Bal = Bal;
    this.Contact = Contact;
}
```

- Here left side fields refer to the fields of current class.

- And right side fields refer to the variables of customer constructor.
- To avoid this kind of ambiguities we use ‘this’ keyword for fields of current class.
- Now let us create object of class.

```
static void Main(string[] args)
{
    Customer c = new Customer(123, "Jack", 78000, "9676010101");

    c.BalEnq();

    Console.ReadLine();
}
```

- Press F5.



- So this is the use of this keyword.
- What is this keyword?
- This keyword refers to the object of current class, in whatever class we using this it acts like its default object.
- Like we are creating an object ‘c’ of class customer in the same way ‘this’ is nothing but the default object of class customer.
- This is the one use.
- There are two uses of this keyword.
- One use is to initialize the object we can use this keyword in assigning the values of variables to the fields of class to avoid ambiguities.
- Before we proceed for another use we need to overload constructor with three parameters.

```
public Customer(int AcNo, string Name, double Bal)
{
    this.AcNo = AcNo;
    this.Name = Name;
    this.Bal = Bal;
}

public Customer(int AcNo, string Name, double Bal,string Contact)
{
    this.AcNo = AcNo;
    this.Name = Name;
    this.Bal = Bal;
    this.Contact = Contact;
}
```

- Why because we may have few customers with no contact number.
- For them instead of passing 'null' while creating an object we'll create another object of customer with no contact number.

```
Customer c1 = new Customer(124, "Peter", 89000);

c1.BalEnq();
```

- Here we've object with no contact number.
- To initialize this object we've following constructor.
- So we can have n number of overloaded customers.
- If we observer above constructors the account number, name and balance is getting repeated.
- What we want to do, already we've written that code and we don't want to write that three lines of code again.
- We wants to use that code which is above.
- Whenever our program gets invoking constructor with four parameter we'll asks them to go up and initialize this three variables here and come down to initialize the last variable.

- That means some here we need to call three parameters constructor explicitly.
- We can achieve this with the help of 'this' keyword.

```
public Customer(int AcNo, string Name, double Bal,string Contact):this(AcNo,Name,Bal)
{
    //this.AcNo = AcNo;
    //this.Name = Name;
    //this.Bal = Bal;
    this.Contact = Contact;
}
```

- Here in 'this' we are passing three parameters to initialize them at three parameterized constructor.
- And remaining parameter i.e. contact is initializing at four parameterized constructor.
- Put a break point to see the details.

```
class Program
{
    static void Main(string[] args)
    {
        Customer c = new Customer(123, "Jack", 78000, "9676010101");

        c.BalEnq();
```

- Now control will jumps to four parameterized constructor.

```
public Customer(int AcNo, string Name, double Bal,string Contact):this(AcNo,Name,Bal)
{
    //this.AcNo = AcNo;
    //this.Name = Name;
    //this.Bal = Bal;
    this.Contact = Contact;
}
```

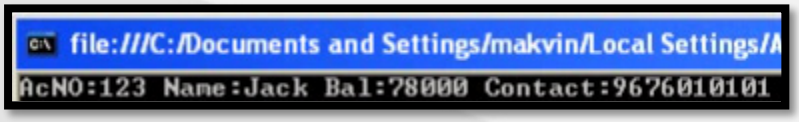
- From here at the occurrence of 'this' keyword with three parameters control jumps to three parameterized constructor to initialize that parameters.

```
public Customer(int AcNo, string Name, double Bal):this()
{
    this.AcNo = AcNo;
    this.Name = Name;
    this.Bal = Bal;
}
```

- After initializing finally control goes back to four parameterized constructor to initialize leftover field.

```
public Customer(int AcNo, string Name, double Bal,string Contact):this(AcNo,Name,Bal)
{
    //this.AcNo = AcNo;
    //this.Name = Name;
    //this.Bal = Bal;
    this.Contact = Contact;
}
```

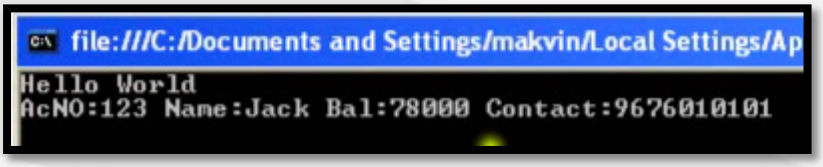
- Now it will call the balance enquiry method.



- So we've seen two uses of this keyword.
- One is to refer the fields of current class.
- And another is to invoke one constructor from the other.
- For example if we mention 'this' keyword with no parameter at three parameterized constructor.
- Simply control jumps from there to no parameter constructor or default constructor.

```
public Customer()
{
    Console.WriteLine("Hello World");
}

public Customer(int AcNo, string Name, double Bal):this()
{
    this.AcNo = AcNo;
    this.Name = Name;
    this.Bal = Bal;
}

public Customer(int AcNo, string Name, double Bal,string Contact):this(AcNo,Name,Bal)
{
    //this.AcNo = AcNo;
    //this.Name = Name;
    //this.Bal = Bal;
    this.Contact = Contact;
}
```

- At the time of execution control first jumps to four parameterized constructor from there control jumps to three parameterized constructor then control jumps to no parameter constructor.
- Press F5.



- See it executes the default constructor first then it executes the three parameterized constructor then finally it executes four parameterized constructor.
- So this is all about 'this' keyword.
- Complete program is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace thisKeyWordEg
```

```
{
    class Customer
    {
        int AcNo;
        string Name;
        double Bal;
        string Contact;

        public Customer()
        {
            Console.WriteLine("Hello World");
        }

        public Customer(int AcNo, string Name, double Bal):this()
        {
            this.AcNo = AcNo;
            this.Name = Name;
            this.Bal = Bal;
        }

        public Customer(int AcNo, string Name, double Bal,string Contact):this(AcNo,Name,Bal)
        {
            this.Contact = Contact;
        }

        public void Deposit(double amt)
        {
            Bal = Bal + amt;
        }
```

```
public void WithDraw(double amt)
{
    Bal = Bal - amt;
}

public void BalEnq()
{
    Console.WriteLine("AcNO:{0} Name:{1} Bal:{2} Contact:{3}", AcNo, Name, Bal, Contact);
}

public void BalEnq(double amt, string f)
{
    if (f == "D")
    {
        Bal += amt;
    }
    else if (f == "W")
    {
        Bal -= amt;
    }
    else
    {
        Console.WriteLine("INVALID Flag");
    }

    Console.WriteLine("AcNO:{0} Name:{1} Bal:{2} Contact:{3}", AcNo, Name, Bal,Contact);
}
```

```
    }

    class Program
    {
        static void Main(string [] args)
        {
            Customer c = new Customer(123, "Jack", 78000, "9676010101");

            c.BalEnq();

            Customer c1 = new Customer(124, "Peter", 89000);

            c1.BalEnq();

            Console.ReadLine();
        }
    }
}
```

CHAPTER 10 : STATIC VARIABLE AND STATIC CONSTRUCTOR.

- In this Chapter we'll see the Static Variable
- We'll take one example class called housing loan.

```
class HousingLoan
{
    int AcNo;
    string Name;
    double LoanAmount;
    double ROI;
}
```

- In this class we've account number of the person who has taken the housing loan and the name, loan amount and the rate of interest that we are applying on the loan amount.
- If we observe if there are n number of customers who has taken housing loan all of them may have different account numbers, names and loan amounts whereas rate of interest would be same for each and every customer.
- So if we create 10 objects or 10 customers then we need to allocate 10 memory locations for ROI.
- And all the 10 location have same value.
- So which results in wastage of memory and another drawback is that if we wants to update ROI we need to update it in all the objects.
- We need to allocate memory for ROI only once and allow all the objects to access it.
- So we just make ROI as static and also make it as public so that we can access it from outside the class to set the value of ROI.

```
class HousingLoan
{
    int AcNo;
    string Name;
    double LoanAmount;
    public static double ROI;
}
```

- Now this three things we need to initialize whenever we need to create the object.
- So for that we'll use constructor following constructor.

```
class HousingLoan
{
    int AcNo;
    string Name;
    double LoanAmount;
    public static double ROI;

    public HousingLoan(int AcNo,
    string Name,
    double LoanAmount)
    {
        this.AcNo = AcNo;
        this.Name = Name;
        this.LoanAmount = LoanAmount;
    }
}
```

- This is our constructor using which we initialize the fields of class.
- And we'll have a method display which is going to display all the details of customer.

```
public void Display()
{
    Console.WriteLine("AcNo:{0} Name:{1} LoanAmount:{2} ROI:{3}",AcNo,Name,LoanAmount,ROI);
}
```

- Now we'll create two objects for housing loan.
- And we'll call display method.

```
HousingLoan h = new HousingLoan(123, "Peter", 56000);
h.Display();

HousingLoan h1 = new HousingLoan(124, "Jack", 78000);
h1.Display();
```

- If we execute it's going to display ROI as 0 because we haven't initialize the ROI.

```
file:///C:/Documents and Settings/makvin/Local Settir
AcNo:123 Name:Peter LoanAmount:56000 ROI:0
AcNo:124 Name:Jack LoanAmount:78000 ROI:0
```

- See it displays ROI as 0.
- So we need to initialize the static variable.
- How do we access the static variable?
- Normally if we want to access the members of class we need to access it with the help of object.
- Whereas static members could be access directly with the help of class name.
- We need not to create object to access static members.

```
HousingLoan.ROI = 12.8;
```

- Press F5.

```
file:///C:/Documents and Settings/makvin/Local Settings/Applic
AcNo:123 Name:Peter LoanAmount:56000 ROI:12.8
AcNo:124 Name:Jack LoanAmount:78000 ROI:12.8
```

- See that both objects has same ROI.
- If we want to update ROI we can use below code.

```
HousingLoan.ROI = 13.8;

h1.Display();
h.Display();
```

- After updating our output will be.

```
file:///C:/Documents and Settings/makvin/Local Settings/
AcNo:123 Name:Peter LoanAmount:56000 ROI:12.8
AcNo:124 Name:Jack LoanAmount:78000 ROI:12.8
AcNo:124 Name:Jack LoanAmount:78000 ROI:13.8
AcNo:123 Name:Peter LoanAmount:56000 ROI:13.8
```

- We can see that ROI is reflected for all the objects.
- Static members has two features one is physical and another is functional.
- Physical feature it looks like static variable declared with keyword static and class can access the static variable directly with the class name objects cannot access the static variables.
- Functional feature is that all the objects would be sharing same memory location for static variable.
- In the display method we'll add one more line of code i.e. repay amount.

```
public void Display()
{
    Console.WriteLine("AcNo:{0} Name:{1} LoanAmount:{2} ROI:{3}",AcNo,Name,LoanAmount,ROI);
    Console.WriteLine("Repay Amount:"+(LoanAmount+(LoanAmount*ROI)/100));
}
```

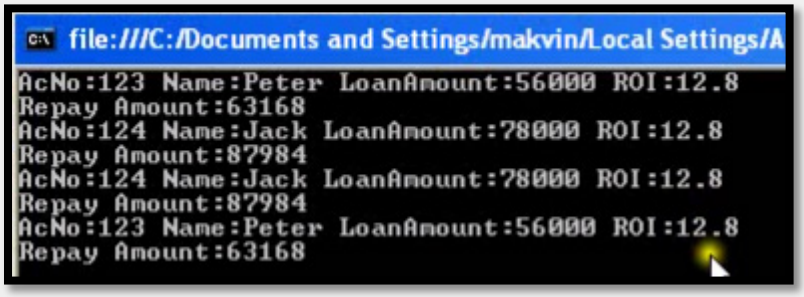
- Press F5.

```
file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temp
AcNo:123 Name:Peter LoanAmount:56000 ROI:12.8
Repay Amount:63168
AcNo:124 Name:Jack LoanAmount:78000 ROI:12.8
Repay Amount:87984
AcNo:124 Name:Jack LoanAmount:78000 ROI:13.8
Repay Amount:88764
AcNo:123 Name:Peter LoanAmount:56000 ROI:13.8
Repay Amount:63728
```

- Now if we've taken loan amount of 56000 at the ROI 12.8 so we need to repay them 63168.
- If ROI gets updated to 13.8 then we need to repay them 63728.
- So this our static variable works.
- Now as per the standards and rules we need to declare the variable as private.
- If we make static variable as private it throws an error in main method it says that we cannot access the private members outside the class.
- What is that we can do?
- We can write a separate constructor where we can initialize static members.
- So we'll write constructor using which we can initialize static variable.
- So that is nothing but our **static constructor**.

```
int AcNo;
string Name;
double LoanAmount;
static double ROI;

static HousingLoan()
{
    ROI = 12.8;
}
```

- In static constructor we can access static variable only.
- We cannot access non static variables inside the static constructor.
- Press F5.



- It has got 12.8.
- All objects share same ROI as 12.8.
- There are few things that's need to note about static constructor.
- It's declared with static keyword.
- Static constructor will not have any return type as like normal constructor.
- It cannot have any kind of access specifiers like public, private and protected.
- Static constructor got executed before creation of any objects.
- Static constructor is parameter less constructor.
- So we cannot overload static constructor.
- We can have one and only one static constructor in the class.
- We can initialize only static variable inside the Static constructor.
- So this is all about static constructor.

CHAPTER 11 : STATIC METHODS AND STATIC METHOD OVERLOADING

- In this chapter we'll see static methods.
- Now we've same example with us that we've done in earlier chapter.
- For example we've some enquiries with us like if anybody comes to the bank and if they want to have an enquiry regarding housing loan.
- So we never create the object for them we never give them the account number.
- We'll just asks how much amount of loan you want.
- Depending upon that loan we'll say ROI and you need to repay so and so amount.
- If we have this kind of scenario where we need to perform some operation where we need not to create the object.
- Without creation of the object we want to perform something.
- For that kind of scenario we can go for **static method**.

```
public static void Enq(double amt, int months)
{
    double repayAmpunt = amt + (amt * ROI) / 100;
    Console.WriteLine("Loan Amount:{0} ROI:{1} RepayAmount:{2} EMI:{3} For {4} months",amt,ROI,repayAmpunt,
}
```

- Now we've static method which takes two parameters.
- One is amount and another is number of months they are going to repay it.
- And it is going to display that things.
- Whenever we want to execute static methods we cannot execute it using objects.
- We can access static methods directly using class name.

```
HousingLoan.Enq(68000, 12);

Console.ReadLine();
```

- Here we've take 68000 loan and we are going to repay it 12 months.

```
Loan Amount:68000 ROI:12.8 RepayAmount:76704 EMI:6392 For 12 months
```

- So this is our static method.
- Static method is a method which is declared with the keyword static.
- We can invoke static method direct with the class name as like static variable.
- We cannot invoke static method with the objects.
- We can use only static members or variables of the class.
- We cannot use non-static members in static methods.
- We can use local variables in Static method.
- Whereas non-static methods can use both static variables and non-static variables.
- We can also overload a static method.
- We can have static method overloading whereas we cannot overload static constructor.

```
public static void Enq(double amt, int months)
{
    double repayAmpunt = amt + (amt * ROI) / 100;
    Console.WriteLine("Loan Amount:{0} ROI:{1} RepayAmount:{2} EMI:{3} For {4} months",amt,ROI,repayAmpunt,
}

public static void Enq(double amt)
{
    double repayAmpunt = amt + (amt * ROI) / 100;
    Console.WriteLine("Loan Amount:{0} ROI:{1} RepayAmount:{2} ", amt, ROI, repayAmpunt);
}
```

- Above code has two same static methods but with different parameters.
- So this is the concept of static method overloading.

- At this time we'll call the method with single parameter i.e. loan amount.

```
HousingLoan.Enq(68000, 12);
HousingLoan.Enq(45000);
```

- Here we've called two same methods but with different parameters.
- Press F5.

```
Loan Amount:68000 ROI:12.8 RepayAmount:76704 EMI:6392 For 12 months
Loan Amount:45000 ROI:12.8 RepayAmount:50760
```

- One more point about static constructor.
- When does the static constructor gets executed?
- Static constructor gets executed before creation of any object.
- And there is no specific time for it.
- But it gets invoke before creation of any object.
- This is all about static method.
- If we observe we have been writing our code in main method which is static.
- So this method is inside the class program.
- If we observe that we are not at all creating an object of program to execute main method.
- Main method gets executed automatically because it is static.
- We've been using many other static methods like WriteLine(), ReadLine() etc.
- See that 'console' is a class WriteLine() is a method.
- Whenever we want to execute the WriteLine() method.
- We never went to create an object of 'console' class then call WriteLine().
- We just calling WriteLine() method directly with the class name.
- So console is class name and WriteLine() is method which is static.

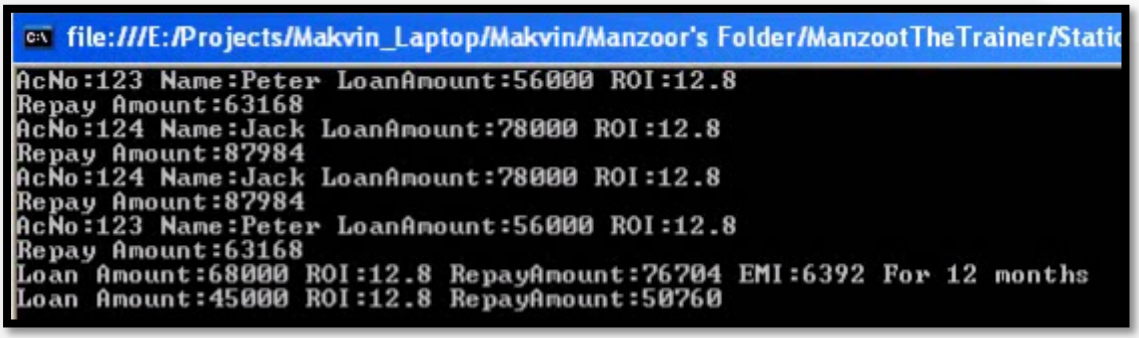
- Like that ReadLine() also a method which is static.
- Clear to our knowledge we have been using static methods.
- Any methods which is accessing directly with the class name is a static method.
- So this is all about static method.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace StaticEg
{
    class HousingLoan
    {
        int AcNo;
        string Name;
        double LoanAmount;
        static double ROI;

        static HousingLoan()
        {
            ROI = 12.8;
        }

        public HousingLoan(int AcNo,
        string Name,
```

```
        double LoanAmount)
        {
            this.AcNo = AcNo;
            this.Name = Name;
            this.LoanAmount = LoanAmount;
        }

        public void Display()
        {
            Console.WriteLine("AcNo:{0} Name:{1} LoanAmount:{2}
ROI:{3}",AcNo,Name,LoanAmount,ROI);
            Console.WriteLine("Repay Amount:"+(LoanAmount+(LoanAmount*ROI)/100));
        }

        public static void Enq(double amt, int months)
        {
            double repayAmpunt = amt + (amt * ROI) / 100;
            Console.WriteLine("Loan Amount:{0} ROI:{1} RepayAmount:{2} EMI:{3} For {4}
months",amt,ROI,repayAmpunt,repayAmpunt/months,months);
        }

        public static void Enq(double amt)
        {
            double repayAmpunt = amt + (amt * ROI) / 100;
            Console.WriteLine("Loan Amount:{0} ROI:{1} RepayAmount:{2} ", amt, ROI, repayAmpunt);
        }

    }
    class Program
    {
```

```
        static void Main(string [] args)
        {
            //HousingLoan.ROI = 12.8;

            HousingLoan h = new HousingLoan(123, "Peter", 56000);
            h.Display();

            HousingLoan h1 = new HousingLoan(124, "Jack", 78000);
            h1.Display();

            //HousingLoan.ROI = 13.8;

            h1.Display();
            h.Display();

            HousingLoan.Enq(68000, 12);
            HousingLoan.Enq(45000);

            Console.ReadLine();
        }
    }
}
```

- Press F5.



- Here is all about static method and static method overloading.
- Thank you..!

CHAPTER 12 : PROPERTIES IS C#.

- In this chapter we'll see what a property is.
- We've one example of class student.

```
class Student
{
    int Sid;
    string SName;
    double m1, m2, m3;

    public Student(int Sid, string SName)
    {
        this.Sid = Sid;
        this.SName = SName;
    }

    public void SetMarks(int m1,int m2,int m3)
    {
        this.m1 = m1;
        this.m2 = m2;
        this.m3 = m3;
    }

    public void DisplayAvg()
    {
        Console.WriteLine("Sid:(0) SName:(1) m1:(2) m2:(3) m3:(4) Avg:(5)",Sid,SName,m1,m2,m3,(m1+m2+m3)/3.0);
    }
}
```

- Class student contain some fields, constructor for initializing the student id and student name but we are not initializing the marks because we need to re initialize the marks for each and every exam so we kept it in separate method for assigning the marks.
- And we've two methods.
- One is for set marks and another is for display average details of student.
- Let us create the object of student and we'll call two methods.

```
Student S = new Student(123, "Peter");
S.DisplayAvg();


S.SetMarks(34, 56, 78);
S.DisplayAvg();
Console.ReadLine();
```

- Press F5.

```
file:///C:/Documents and Settings/makvin/Local Settings/
Sid:123 SName:Peter m1:0 m2:0 m3:0 Avg:0
Sid:123 SName:Peter m1:34 m2:56 m3:78 Avg:56
```

- Our average is 56.
- Our requirement is we need to set the marks of each subject separately.
- Another thing is to get the marks of each subject separately.
- We don't want to display the details over here.
- We want to get the value back to our program.
- What is that we need to do?
- As a method can return a single value.

```
public double Getm1()
{
    return m1;
}
```

- This is going to return marks m1.
- Whenever we want to get the marks m1 we need to call that method.

```
Console.WriteLine("M1:" + S.Getm1());
```

```
file:///C:/Documents and Settings/makvin/Local Settings
Sid:123 SName:Peter m1:0 m2:0 m3:0 Avg:0
Sid:123 SName:Peter m1:34 m2:56 m3:78 Avg:56
M1:34
```

- This going to return m1.
- Till now we've seen methods which were not returning any values so we were using 'void' return type.
- This method returns double so our return type is double and it is going return m1.

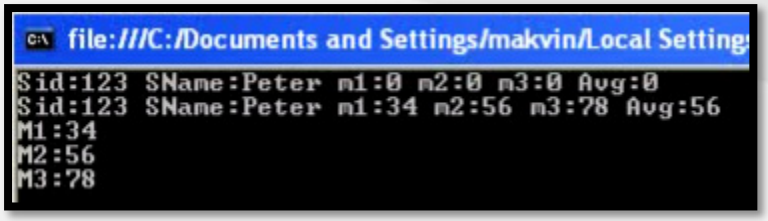
- So that value we are using in main method.
- Now if we want to use m2 so we need to write one more method.
- For m3 we need to write one more method.

```
public double Getm1()
{
    return m1;
}

public double Getm2()
{
    return m2;
}

public double Getm3()
{
    return m3;
}
```

- Same methods we need to call from main.

```
Console.WriteLine("M1:" + S.Getm1());
Console.WriteLine("M2:" + S.Getm2());
Console.WriteLine("M3:" + S.Getm3());
```

- Press F5.



- Here we got m1, m2 and m3.
- Just to have free of accessing the variables so we need to write separate methods for them.
- These are the methods to get.
- Now we want to set single marks m1 so we need to write separate method.

```
public void Setm1(double m1)
{
    this.m1 = m1;
}
```

- Whenever we want to set only marks m1 we'll just call s.setm1 () method.

```
S.Setm1(67);
```

- After setting m1 we'll just go for calculating the average.

```
file:///C:/Documents and Settings/makvin/Local Settings
Sid:123 SName:Peter m1:0 m2:0 m3:0 Avg:0
Sid:123 SName:Peter m1:34 m2:56 m3:78 Avg:56
M1:34
M2:56
M3:78
Sid:123 SName:Peter m1:67 m2:56 m3:78 Avg:67
```

- Here new m1 is 67 and average is 67.
- To set m2 and m3 we need to write two separate set methods.
- But toughest thing over here is that we need to call the method each and every time to perform simple operation.
- To access simple variable we need to call a method.

```
public void Setm1(double m1)
{
    this.m1 = m1;
}
public void Setm2(double m2)
{
    this.m2 = m2;
}
public void Setm3(double m3)
{
    this.m3 = m3;
```

- Here we have three set methods separately and three get methods.
- And each and every time we need to call the method to get the value.

- And we need to call the method to set the value.
- To enhance this we can go for **properties**.
- Means what our variables are private whereas we'll define a property that will be public for the same variable.
- In this we'll have two methods.
- One is set and another is get.

```
public double M1
{
    set { m1 = value; }
    get { return m1; }
}
```

- In set method we've value is a default variable which will store the value whatever we going to assign.
- And get method returns m1.
- This is our simple m1 property works like setm1 and getm1 methods.
- In this property we've one setter method and a getter method.
- Whenever we want to assign the value of m1 we can simple write s.M1=67 to set 67 for marks m1.
- Like that we'll create properties for m2 and m3.

```
public double M2
{
    set { m2 = value; }
    get { return m2; }
}

public double M3
{
    set { m3 = value; }
    get { return m1; }
}
```

- So we've properties for fields.

- So we'll accessing properties in a way that it looks like we are accessing fields.
- But this properties are intern working on fields.
- So we've got three properties.
- Whenever we want to set the marks instead of calling separate methods we can go for simple operation.
- To access the marks simply we can access like fields.

```
S.M1 = 67;
S.M2 = 78;
S.M3 = 58;

Console.WriteLine("M1:" + S.M1);
Console.WriteLine("M2:" + S.M2);
Console.WriteLine("M3:" + S.M3);
```

- Now we'll go for calling display average method.
- Press F5.

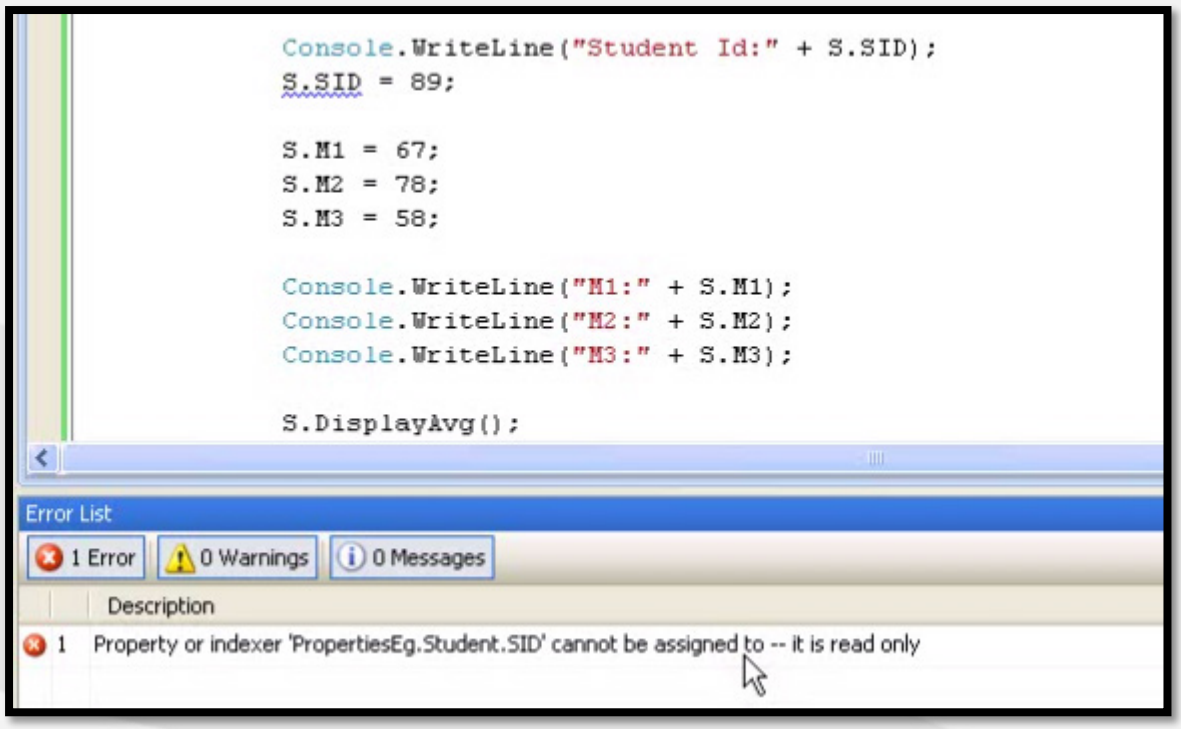
```
M1:67
M2:78
M3:58
Sid:123 SName:Peter m1:67 m2:78 m3:58 Avg:67.6666666666667
```

- So this are the best practices whenever we go for fields it is good to define the properties of each and every fields.
- Now we have many situations where we need to access student id.
- So for student id we can also define the property.

```
public int SID
{
    get { return Sid; }
}
```

- If we observe this we do not have a set method in this.

- Because we don't want to re initialize the id we want to access it.
- It gives us read only permission so we can only read student id.
- We can only access student id for display purpose.
- If we want to write or store the value in SID then it will throw an error.



- Because we have only getter method we do not have setter method.
- If we want to make a property as read only simply we need to remove set method then it becomes read only property.
- So this is all about properties.
- Complete code is given below.

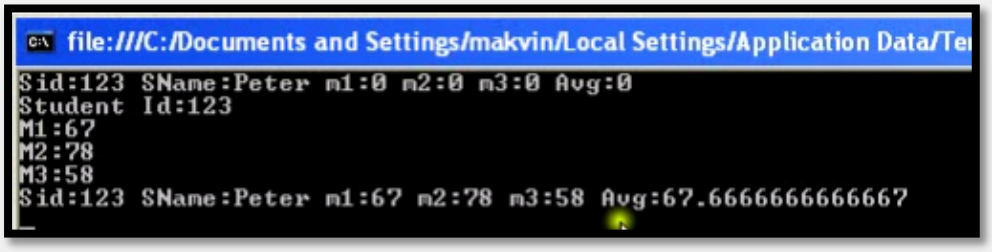
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace PropertiesEg
{
    class Student
    {
        int Sid;
        string SName;
        double m1, m2, m3;

        public int SID
        {
            get { return Sid; }
        }

        public double M1
        {
            set { m1 = value; }
            get { return m1; }
        }

        public double M2
        {
            set { m2 = value; }
            get { return m2; }
        }

        public double M3
        {
            set { m3 = value; }
            get { return m3; }
```

```
}

public Student(int Sid, string SName)
{
    this.Sid = Sid;
    this.SName = SName;
}

public void SetMarks(int m1,int m2,int m3)
{
    this.m1 = m1;
    this.m2 = m2;
    this.m3 = m3;
}

//public double Getm1()
//{
//    return m1;
//}

//public double Getm2()
//{
//    return m2;
//}

//public double Getm3()
//{
//    return m3;
//}
```

```
        //public void Setm1(double m1)
        //{
        //    this.m1 = m1;
        //}
        //public void Setm2(double m2)
        //{
        //    this.m2 = m2;
        //}
        //public void Setm3(double m3)
        //{
        //    this.m3 = m3;
        //}

        public void DisplayAvg()
        {
            Console.WriteLine("Sid:{0} SName:{1} m1:{2} m2:{3} m3:{4}
Avg:{5}",Sid,SName,m1,m2,m3,(m1+m2+m3)/3.0);
        }
    }

    class Program
    {
        static void Main(string [] args)
        {
            Student S = new Student(123, "Peter");
            S.DisplayAvg();

            Console.WriteLine("Student Id:" + S.SID);
```

```
            S.M1 = 67;
            S.M2 = 78;
            S.M3 = 58;

            Console.WriteLine("M1:" + S.M1);
            Console.WriteLine("M2:" + S.M2);
            Console.WriteLine("M3:" + S.M3);

            S.DisplayAvg();

            //S.SetMarks(34, 56, 78);
            //S.DisplayAvg();

            //Console.WriteLine("M1:" + S.Getm1());
            //Console.WriteLine("M2:" + S.Getm2());
            //Console.WriteLine("M3:" + S.Getm3());

            //S.Setm1(67);

            //S.DisplayAvg();

            Console.ReadLine();
        }
    }
}
```



CHAPTER 13 : NAMESPACES IN C#.

- In this chapter we'll see what a namespace is.
- Namespace is nothing but it is a logical collection of classes or aggregation of classes.
- A namespace can have n number of classes as well as sub namespaces.
- We need namespace to avoid naming conflicts.
- For example there is team A, who's working in India and they thought that name of the class is student and they created a class called student.
- In that they implemented the method Display().
- And team B which is working in US they thought that the name of the class for their requirements of the same project may be it suits well is again same student.
- Due to lack of communication or as per the requirements the student name for the class suits well.
- Even they created a method show inside the class.

```
class Student
{
    public void Display()
    {
    }
}

class Student
{
    public void Show()
    {
    }
}
```

- Both of the team drop their codes and they try to integrate but at the time of creating an object of class student it throws an error.

- As the namespace already contains a definition for student.
- That means we cannot have two classes with same name.
- Solution for this is we can have above two different classes in two different namespaces.

```
namespace TeamA
{
    class Student
    {
        public void Display()
        {
        }
    }
}

namespace TeamB
{

    class Student
    {
        public void Show()
        {
        }
    }
}
```

- Here we have TeamA and TeamB namespaces for two different classes of student.
- Now at the time of creation of object we need to use 'TeamA.Student' instead of 'Student'.
- Means at the time of creation of object we've to use namespace name before class name.

```
static void Main(string[] args)
{
    TeamA.Student S = new TeamA.Student();
}
```

- Here we've created an object S of class student which is present in TeamA.

- Like that we can create the object of team B but we need to use 'TeamB.Student'.
- Using this object we can call the method of student.

```
static void Main(string[] args)
{
    TeamA.Student S = new TeamA.Student();
    TeamB.Student S1 = new TeamB.Student();
    S.Display();
    S1.Show();
}
```

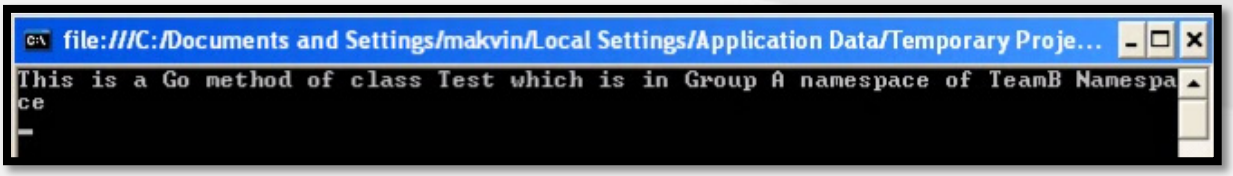
- Here we've created an object S1 of class student which is present in TeamB.
- So the object of class which is present in TeamA calls the method of that particular class.
- Object of class which is present in TeamB calls the method of that particular class.
- Now there is no naming conflicts.
- We have separate namespaces in which we can have same classes.
- We can have same classes but it should be in different namespaces.
- So namespace is nothing but collection of classes.
- We can have namespace inside the namespace.
- Now we'll create another class teacher inside the namespace TeamB.
- And we'll create method GetSalary() inside that class.
- Inside TeamB we can have one more namespace.
- So we we'll create another namespace GroupA inside the TeamB in which we'll create a class 'Test' in this we've a method called as 'Go'.
- In that method we'll try to display a message.

```
namespace TeamB
{

    class Student
    {
        public void Show()
        {
        }
    }

    class Teacher
    {
        public void GetSalary()
        {
        }
    }

    namespace GroupA
    {
        class Test
        {
            public void Go()
            {
                Console.WriteLine("This is a Go method of class Test which is in Group A namespace of TeamB Namespa
            }
        }
```

- Here we've created Group A namespace inside the Team B namespace.
- And it contains the class Test and inside that class we've Go method.
- If we want to create the object of Test again we have to use TeamB.GroupA.Test objectName.

```
TeamB.GroupA.Test t = new TeamB.GroupA.Test();

t.Go();
```

- Press F5.


file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
This is a Go method of class Test which is in Group A namespace of TeamB Namespa
ce

- It displays the message.
- This is the content of Go method.
- If we observe that whenever we want to create an object we need to remember and write the complete path of the class.
- Instead of writing complete path we can simply write that path in using.

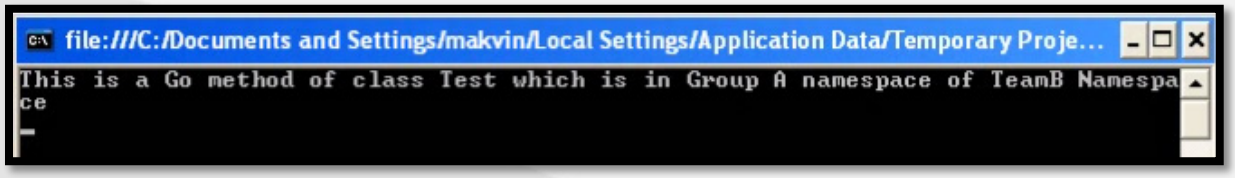
- If we write TeamB.GroupA in using as below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TeamB.GroupA;
```

- Then instead of writing complete path of the class we can simply write Test to create an object as below.

```
Test t = new Test();

t.Go();
```

- Press F5.


file:///C:/Documents and Settings/makvin/Local Settings/Application Data/Temporary Proje...
This is a Go method of class Test which is in Group A namespace of TeamB Namespa
ce

- Its working fine.
- So if you observe we have been writing using System since our first program because System is namespace which contains the class 'console'.
- If we remove this System from using it thrown an error.
- As the name 'console' does not exist in the current context.
- To remove this we can use system namespace before console as below.

```
System.Console.ReadLine();
```

- And our every program is in a namespace.
- By default whatever we project creates it takes that name as the name of the namespace.
- All together namespace is nothing but collection of classes and namespaces.

- And it is use to avoid naming conflicts for the classes and to have good understanding of codes developed by teams of particular organization.
- So this is all about namespaces.
- Complete code is given below.

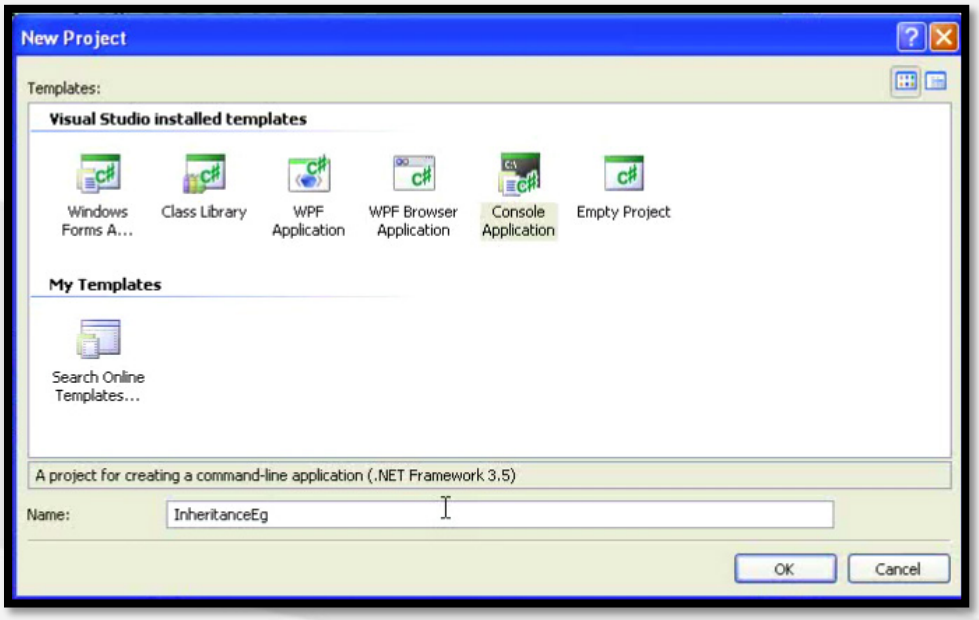
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using TeamB.GroupA;



//namespace TeamA
//{
//    class Student
//    {
//        public void Display()
//        {
//        }
//    }
//}

namespace TeamB
{

    class Student
    {
        public void Show()
        {
        }
```

```
    }


    class Teacher
    {
        public void GetSalary()
        {
        }
    }


    namespace GroupA
    {
        class Test
        {
            public void Go()
            {
                System.Console.WriteLine( "This is a Go method of class Test which is in Group A
namespace of TeamB Namespace " );
            }
        }
    }
}


namespace NameSpaceEg
{
    class Program
    {
        static void Main(string [] args)
        {
            //TeamA.Student S = new TeamA.Student();
```

```
            //TeamB.Student S1 = new TeamB.Student();
            //S.Display();
            //S1.Show();

            //TeamB.GroupA.Test t = new TeamB.GroupA.Test();

            Test t = new Test();

            t.Go();
            System.Console.ReadLine();
        }
    }
}
```

CHAPTER 14 : UNDERSTANDING THE CONCEPT OF INHERITANCE AND PROTECTED VARIABLES IN C#.

- In this chapter we'll see the implementation of Inheritance.
- File→New project→Console application.
- Name it as InheritanceEg.



- We'll see how to implement inheritance and one more access specifier i.e. protected and various types of inheritance.
- Now we'll simulate this with the help of an example class A.
- In this class we've two variables.
- One is int x, and another is public int y.
- As usual if we create the object of class A then definitely we can access the public variables whereas we cannot access the private variables from outside the class.

```
class A
{
    int x;
   public int y;
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();
        //Can Access   a.y;
        //Cannot Access   a.x;
    }
}
```

- So this are the basic access specifiers public and private.
- By default members of the class are private or we can use private keyword explicitly.
- Now for example we've one more class B.
- In this we've again two variables one is private and another is public.
- In the same sense like class A we can create the object of class B.

```
class B
{
    private int l;
    public int m;
}
```

```
B b = new B();
// Can Access b.m;
// Cannot Access b.l;
```

- These are the simple classes we have.
- Now what we want, we want to inherit class A into class B.
- So how do we go for inheritance, we can simply use 'class B: A'.

```
class A
{
    private int x;
    public int y;
}

class B:A
{
     private int l;
     public int m;
}
```

- This means that class B inherits class A.
- And our class B becomes the 'Derived class' or 'child class'.
- And our class A becomes the 'Base class' or 'super class'.
- As well as class B inherit all the members of the class A which are public in this.
- Automatically we get the public int y in class B because class B inherits class A and class A contains public variable.
- Now we can access 'b.y' in main.
- So we can access all the public variables of class A with the help object of class B because class B inheriting class A.
- So this is our simple inheritance.
- Now our main intention behind inheritance is to take all the members of base class into our child class.
- If we observe we can take only public variables of base class into child class.
- We cannot take private variables.
- That means we cannot access it in main method also because if the variable is private we cannot access it with the help of object of same class.
- So here there is no question of accessing the private variable with the object of derived class.

- If we want to variable x as well we should make it as public.
- Now the problem with the public variables is that as per the rules of defining the class all the fields must be private.
- But if we put them as private we cannot inherit them.
- If we make them as public then we can inherit them but there is no security.
- We want the security as well as inheritance.
- For that we'll be using the variable called as 'protected'.
- We'll create a protected variable z.
- Now z gets inherited into the class B.

```
//Base Class
class A
{
    private int x;
    public int y;
    protected int z;
}

//Drived class
class B:A
{
    //protected int z;
    //public int y;
    private int l;
    public int m;

    public void Show()
    {
        //Can Access m
        //Can Access l
        //Can Access y
        //Cannot Access x
        //Can Access z
    }
}
```

```
static void Main(string[] args)
{
    A a = new A();
    //Can Access    a.y;
    //Cannot Access    a.x;
    //Cannot Access a.z


    B b = new B();
    // Can Access b.m;
    // Cannot Access b.l;
    // Can Access b.y;
    // Cannot Access b.x
    // Cannot Access b.z
}
```

- From this example we can say that if a class is standalone class protected and private variables are one and the same things.
- We cannot access them from outside the class.
- If we going for inheritance protected variables get inherited that means we'll have even protected variable z in class B.
- Protected variables get inherited and public variables get inherited whereas private variables will never get inherited into the derived class.
- And we cannot access protected variables from outside the class means full protection.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace InheritanceEg
{
    //Base Class
    class A
    {
```

```
    private int x;
    public int y;
    protected int z;
}

//Drived class
class B:A
{
    //protected int z;
    //public int y;
    private int l;
    public int m;

    public void Show()
    {
        //Can Access m
        //Can Access l
        //Can Access y
        //Cannot Access x
        //Can Access z
    }
}

class Program
{
    static void Main(string [] args)
    {
        A a = new A();
        //Can Access   a.y;
```

```
            //Cannot Access   a.x;
            //Cannot Access a.z

            B b = new B();
            // Can Access b.m;
            // Cannot Access b.l;
            // Can Access b.y;
            // Cannot Access b.x
            // Cannot Access b.z
        }
    }
}
```

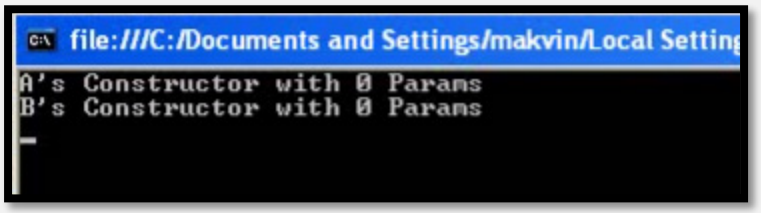
- If we observe the code private variables provides the security but not gets inherited.
- Public variables get inherited but not provides the security.
- Solution to this problem is protected variables.
- Protected variables get inherited as well as provides the full security.

CHAPTER 15 : CONSTRUCTOR CHAINING IN C#.

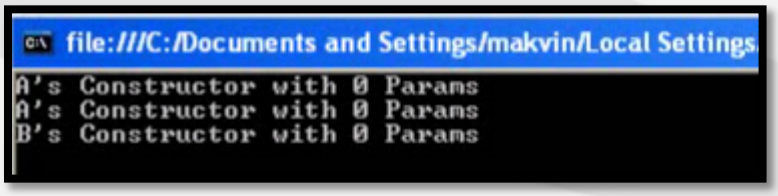
- In this chapter we'll see the concept of constructor chaining.
- We'll start a new console application.
- Now for example we've class A.
- Constructor is a special method, no return type, and same name as class name, and it is invoked automatically whenever we create the object and it is usually public as we have seen in earlier concepts.
- Constructor without any parameter called as 'default constructor'.
- We'll see the constructor chaining concept in detail with the following program.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }
}

class B
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();

        B b = new B();

        Console.ReadLine();
    }
```

- Here we've two default constructors of class A and class B.
- And we've also created an object for that classes.
- Press F5.



- This is the simple concept that we've seen in earlier.
- Now we'll inherit class A into class B simple like **Class B: A.**
- If we execute we assume or our expectation is that our output will be same as earlier because we are creating the object of class A so it is going to invoke A's constructor.
- And we are creating the object of class B so it is going to invoke B's constructor.
- Press F5 and see the output.


file:///C:/Documents and Settings/makvin/Local Settings
A's Constructor with 0 Params
A's Constructor with 0 Params
B's Constructor with 0 Params

- So if we observe the A's constructor got invoked twice.
- How and when, let us see this with a break point.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }
}

class B:A
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }
}

class Program
{
    static void Main(string[] args)
    {
        A a = new A();

        B b = new B();

        Console.ReadLine();
    }
```

- Press F11.
- Now whenever we are trying to create the object of class A it is going to invoke its constructor its fair enough.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }
}
```

- Now we are creating the object of class B so what is our expectation it is going to invoke B's constructor.

```
class B:A
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }
}
```

- Its fine but at the time of invocation of B's constructor it sees that B is inherited from class A.
- Now it becomes the responsibility of class B to invoke its base class constructor.
- So our control will jumps from constructor B to constructor of base class.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }
}
```

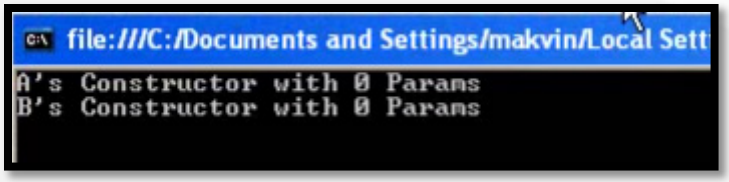
- So it is going to execute A's constructor first then constructor of class B.

```
file:///C:/Documents and Settings/makvin/Local Settings
A's Constructor with 0 Params
A's Constructor with 0 Params
B's Constructor with 0 Params
```

- So if we comment out the A's object.

```
//A a = new A();

B b = new B();
```

- Then we'll get the output as.



- Now what is happening whenever an object of derived class is created it is going to invoke the constructor of its immediate base class.
- The responsibility of derived class is to invoke the constructor of its immediate base class.
- This concept is called as **Constructor Chaining.**
- Now this kind of inheritance known as single inheritance.
- That means class B is inheriting the class A.
- Now we'll create the multi-level inheritance.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }
}

class B:A
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }
}

class C : B
{
    public C()
    {
        Console.WriteLine("C's Constructor with 0 Params");
    }
}
```

- Now what we'll do instead of creating the object of class B we'll simply create the object of class c.

```
//A a = new A();

//B b = new B();

C c = new C();
```

- Let us examine what will happen to the chain of constructor.
- Now we'll just put a break point and see the execution flow.
- Now it is going to jump constructor of class C.

```
class C : B
{
    public C()
    {
        Console.WriteLine("C's Constructor with 0 Params");
    }
}
```

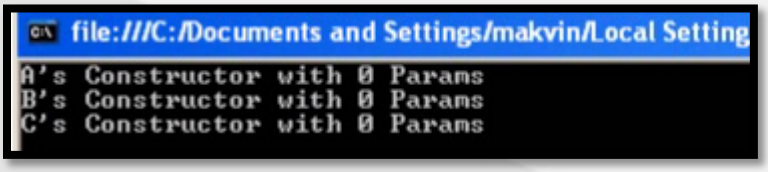
- Now as per the constructor chaining rule it is the responsibility of derived class to invoke its immediate base class constructor.
- So our control will jumps to the constructor of class B.

```
class B:A
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }
}
```

- Now it says that B is again inheriting a new class A.
- So again it becomes is the responsibility of derived class to invoke its immediate base class constructor.
- So our will again jumps to the constructor of class A.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }
}
```

- Then it going to execute A's constructor.
- Then it comes back and executes the B's constructor.
- Then the final C's constructor.
- So whenever we create the object of class C which is inherited from class B which is intern inherited from class A.
- So we'll get A's constructor, B's constructor and C's constructor with 0 Params.


file:///C:/Documents and Settings/makvin/Local Setting
A's Constructor with 0 Params
B's Constructor with 0 Params
C's Constructor with 0 Params

- So this concept is called as **Constructor Chaining.**
- Now here we have constructor with no parameters or default constructor.
- For example if we have a constructor with single parameter if we go for constructor overloading concept.
- In class B we've another constructor with two parameters.
- In class C again we've another constructor with single parameter.

```
class B:A
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }

    public B(int x,int y)
    {
        Console.WriteLine("B's Constructor with 2 Params");
    }
}

class C : B
{
    public C()
    {
        Console.WriteLine("C's Constructor with 0 Params");
    }
    public C(int x)
    {
        Console.WriteLine("C's Constructor with 1 Params");
    }
}
```

- If we execute this it shows same output as earlier because our derived class constructor going to invoke its immediate base class default constructor.

```
file:///C:/Documents and Settings/makvin/Local Setting
A's Constructor with 0 Params
B's Constructor with 0 Params
C's Constructor with 0 Params
```

- But what we want if our control comes to constructor of class C, we want it to jumps to two parameter constructor instead of jumping to the constructor of no parameter.
- We can do that with the help of a keyword called as **base**.
- So in base we can pass two parameters to jump it to two parameter constructor.
- Now we'll put a breakpoint and let us see what happens.

```
class C : B
{
    public C():base(2,4)
    {
        Console.WriteLine("C's Constructor with 0 Params");
    }
    public C(int x)
    {
        Console.WriteLine("C's Constructor with 1 Params");
    }
}
```

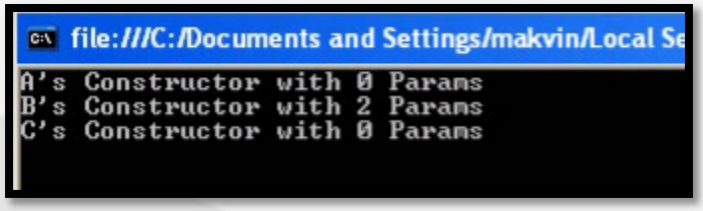
- As per the normal rules our class C object will automatically going to invoke C's default constructor.
- From here in our earlier example our control was jumping to default constructor of base class.
- But now we are saying that base with two parameters that means we are explicitly calling the base class constructor with two parameters.
- So our control should jumps from here to two parameter constructor of base class.

```
class B:A
{
    public B()
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }

    public B(int x,int y)
    {
        Console.WriteLine("B's Constructor with 2 Params");
    }
}
```

- Now here we are not calling any constructors of base class explicitly.
- So by default it is going to jump to the default constructor of base class.

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }

    public A(int x)
    {
        Console.WriteLine("A's Constructor with 1 Params");
    }
}
```

- So our control first executes this then comes back to parameterized constructor of class B then our default constructor of class C.


file:///C:/Documents and Settings/makvin/Local Se
A's Constructor with 0 Params
B's Constructor with 2 Params
C's Constructor with 0 Params

- It displays first default constructor of class A then two parameter constructor of class B and then finally constructor of class C.
- So what is base?
- **Base** is a keyword which is used to invoke one constructor from another of base class from the derived class.
- We can invoke constructor of the base class from the derived class with the help of base keyword.
- Now we'll be creating the object of class C with single parameter.

```
public C(int x)
{
    Console.WriteLine("C's Constructor with 1 Params");
}
```

- So it has invoke the constructor of single parameter.

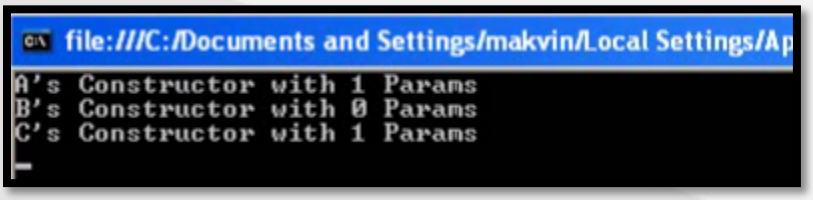
- Now the job of derived class to invoke its immediate base class constructor default constructor.

```
public B():base(5)
{
    Console.WriteLine("B's Constructor with 0 Params");
}
```

- But from here we are explicitly calling base class constructor with single parameter.

```
public A(int x)
{
    Console.WriteLine("A's Constructor with 1 Params");
}
```

- So it first executes the single parameter constructor of class A, next default constructor of class B and finally single parameter constructor of class C.


file:///C:/Documents and Settings/makvin/Local Settings/Ap
A's Constructor with 1 Params
B's Constructor with 0 Params
C's Constructor with 1 Params

- In this way we can change the constructor chaining as per our requirements.
- This is all about constructor chaining, base keyword and multi-level inheritance.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace ConstructorChainingEg
{
```

```
class A
{
    public A()
    {
        Console.WriteLine("A's Constructor with 0 Params");
    }

    public A(int x)
    {
        Console.WriteLine("A's Constructor with 1 Params");
    }
}

class B:A
{
    public B():base(5)
    {
        Console.WriteLine("B's Constructor with 0 Params");
    }

    public B(int x,int y)
    {
        Console.WriteLine("B's Constructor with 2 Params");
    }
}

class C : B
{
    public C():base(2,4)
```

```
        {
            Console.WriteLine("C's Constructor with 0 Params");
        }
        public C(int x)
        {
            Console.WriteLine("C's Constructor with 1 Params");
        }
    }

    class Program
    {
        static void Main(string [] args)
        {
            //A a = new A();

            //B b = new B();

            C c = new C(5);

            Console.ReadLine();
        }
    }
}
```

CHAPTER 16 : METHOD OVERRIDING, VARIOUS TYPES OF INHERITANCE AND CONSTRUCTOR CHAINING IN C#.

- In this chapter we are going to see what is method overriding?
- Now before we proceed for method overriding let us implement all the things that we've learnt till now in inheritance.
- For example we've a class called as student.

```
class Student
{
    protected int Sid;
    protected string Name;
    protected string Standard;

    public Student(int Sid, string Name, string Standard)
    {
        this.Sid = Sid;
        this.Name = Name;
        this.Standard = Standard;
    }

    protected double Maths, Science, Social;

    public double _Maths
    {
        set { Maths = value; }
        get { return Maths; }
    }
```

```
    public double _Science
    {
        set { Science = value; }
        get { return Science; }
    }

    public double _Social
    {
        set { Social = value; }
        get { return Social; }
    }
```

```
public void GetDetails()
{
    Console.WriteLine("SID : " + Sid);
    Console.WriteLine("Name : " + Name);
    Console.WriteLine("Standard : " + Standard);
}
```

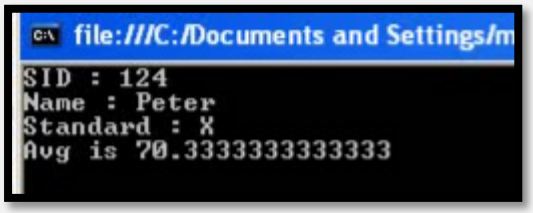
```
public void GetAvg()
{
    Console.WriteLine("Avg is " + (Maths + Science + Social) / 3.0);
}
```

- And we are declaring all the fields as protected because we want to inherit in future.
- To initialize fields we've constructor.
- Now the student has got 3 subjects and the marks of the 3 subjects need to be assign.
- So we've used properties that we've learnt in earlier chapters.
- We've the properties for 3 subjects.
- And we've a method GetDetails which displays SID, Name and Standard.
- And a method GetAvg which displays the average.
- So this is our simple class.
- Now we'll create the object of this class.

```
static void Main(string[] args)
{
    Student s = new Student(124, "Peter", "X");
    s._Maths = 67;
    s._Science = 78;
    s._Social = 66;
    s.GetDetails();
    s.GetAvg();
    Console.ReadLine();
}
```

- So this is our simple class with two methods and three properties.

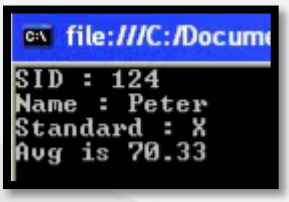
- Press F5.



- We can round the result with math class.

```
public void GetAvg()
{
    Console.WriteLine("Avg is " +Math.Round((Maths + Science + Social) / 3.0,2));
}
```

- We rounded to two decimals.



- For example if we give this application to any school it works fine.
- If some school want same details in addition to this they require to add one more subject, and maintain contact no. and add one more method for division.
- They have three extra requirements.
- Means we've to extend the functionality of existing class.
- So we can go for inheritance.
- So we'll be creating new student class which inherits old student class.
- Once we go for inheritance all the public and protected members of old student class will get inherited into new student class.
- Now we can add new fields in derived class that is required by new school.

```
class NewStudent : Student
{
    protected string Contact;

     public NewStudent(int Sid, string Name, string Standard,string Contact)
    {
        'MethodOverridingEg.Student' does not contain a constructor that takes '0' arguments
        this.Sid = Sid;
        this.Name = Name;
        this.Standard = Standard;
        this.Contact = Contact;
    }

}
```

- Now if we observe there is an error at constructor.
- Its showing there is no default constructor in base class.
- But as per constructor chaining there is responsibility of derived class constructor is to invoke its immediate base class default constructor.
- To avoid this we need to create default constructor for student class.

```
public Student()
{
}
```

- Whenever we don't have any constructor in the class then we need not to write default constructor it will take automatically.
- Whenever we have constructor with parameter then we must need to create default constructor.
- One requirement got fulfilled.
- Now next requirement is they need one more subject.

```
protected double English;

public double _English
{
    set { English = value; }
    get { return English; }
}
```

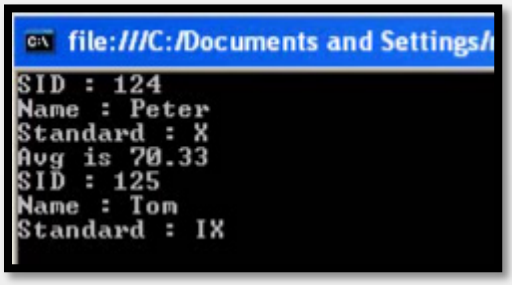
- Two requirement we've fulfilled.
- And last requirement is division method.

```
public void GetDivision()
{
    double avg = (Maths + Science + Social + English) / 4.0;
    if (avg < 35)
        Console.WriteLine("Fail");
    else if(avg<60)
        Console.WriteLine("Pass");
    else if(avg<70)
        Console.WriteLine("Passed in first Division");
    else if(avg<=100)
        Console.WriteLine("Passed in Dist");
}
```

- Now we'll create an object of class new student.
- And with this object we can access all the methods because it is derived class object.

```
NewStudent ns = new NewStudent(125, "Tom", "IX", "9676010101");
ns._English = 77;
ns._Maths = 89;
ns._Science = 55;
ns._Social = 98;

ns.GetDetails();
```

- We'll just execute this.
- Now press F5.



- See it showing tom details.

- If we observe in tom details it does not showing the contact number because in our NewStudent class we do not have GetDetails method.
- It is using GetDetails method of our parent class.
- So we need to write GetDetails method in derived class too.
- In GetDetails method we need to add one more line of code i.e. contact.

```
public void GetDetails()
{
    Console.WriteLine("SID : " + Sid);
    Console.WriteLine("Name : " + Name);
    Console.WriteLine("Standard : " + Standard);
    Console.WriteLine("Contact : " + Contact);
}
```

- If we observe our base class and derived class has same GetDetails method with no parameters and same return type.
- This complete thing is called as **signature** i.e. public void GetDetails().
- Signature of the method in base class is same as signature of the method in derived class.
- But the implementation is different.
- Whenever we've a method with same signature in the derived class as same as in base class with different implementation then we say the method of derived class overrides the method of base class.
- It means whenever we call GetDetails it executes derived class method it is not at all executes base class method.
- This concept is called as **method overriding**.
- Same thing we need to implement for GetAvg method because previous method was calculating average for 3 subjects.
- But now we've 4 subjects.

```
public void GetAvg()
{
    Console.WriteLine("Avg is " + Math.Round((Maths + Science + Social + English) / 4.0, 2));
}
```

- So we've overridden even this GetAvg method.
- We can see that method in the derived class has same signature as that of the base class signature but with different implementation.
- This process is called as method overriding.
- But one thing we need to take care whenever we want to override a method we need to declare the method of the base class as **virtual** and method of the derived class as **override**.

```
public virtual void GetDetails()
{
    Console.WriteLine("SID : " + Sid);
    Console.WriteLine("Name : " + Name);
    Console.WriteLine("Standard : " + Standard);
}
```

```
public override void GetDetails()
{
    Console.WriteLine("SID : " + Sid);
    Console.WriteLine("Name : " + Name);
    Console.WriteLine("Standard : " + Standard);
    Console.WriteLine("Contact : " + Contact);
}
```

- **Override** is a keyword which is use to override a method of the base class which is declared with the keyword virtual.
- If we execute GetAvg method it executes derived class method it is not at all going to executes the base class method.
- Now let us call remaining methods.

```
NewStudent ns = new NewStudent(125, "Tom", "IX", "9676010101");
ns._English = 77;
ns._Maths = 89;
ns._Science = 55;
ns._Social = 98;

ns.GetDetails();
ns.GetAvg();
ns.GetDivision();
```

- Press F5.

```
SID : 125
Name : Tom
Standard : IX
Contact : 9676010101
Avg is 79.75
Passed in Dist
```

- Now it's displaying all the details.
- So we have successfully use the feature of inheritance and we've implemented a new class student.
- One more thing if we observe whenever we creating the object constructor gets invoked.
- Now it is the responsibility of derived class constructor to invoke its base class constructor so this is the feature we've seen in earlier chapter.
- If we observe that there is code gets repeated in derived class.
- So we use this constructor chaining feature.
- That means from derived class constructor we'll call constructor of base class with three parameters.
- So we can do this with 'base' keyword.
- We'll use base keyword at derived class constructor and pass three parameters i.e. SID, Name and Standard.

```
public NewStudent(int Sid, string Name, string Standard,string Contact):base(Sid,Name,Standard)
{
    //this.Sid = Sid;
    //this.Name = Name;
    //this.Standard = Standard;
    this.Contact = Contact;
}
```

- So that these three variables will get initialized at base class constructor of three parameters.
- And remaining one parameter get initialized here.
- So in this way we can use the concept of constructor chaining.
- So this is the use of base keyword.
- One more thing if we observe GetDetails method in the derived class there is three lines of code is same as that we have in base class.

```
public override void GetDetails()
{
    Console.WriteLine("SID : " + Sid);
    Console.WriteLine("Name : " + Name);
    Console.WriteLine("Standard : " + Standard);
    Console.WriteLine("Contact : " + Contact);
}
```

- So instead of writing this three lines of code again and again we can simply call base class method from here.
- We can call base class method as below.

```
public override void GetDetails()
{
    //Console.WriteLine("SID : " + Sid);
    //Console.WriteLine("Name : " + Name);
    //Console.WriteLine("Standard : " + Standard);
    base.GetDetails();
    Console.WriteLine("Contact : " + Contact);
}
```

- So this three things it is executing from there and left out part is executing from here.

- Put a breakpoint and see the flow.

```
ns.GetDetails();
ns.GetAvg();
ns.GetDivision();
```

- From here our control will jumps to the GetDetails method of NewStudent class.

```
public override void GetDetails()
{
    //Console.WriteLine("SID : " + Sid);
    //Console.WriteLine("Name : " + Name);
    //Console.WriteLine("Standard : " + Standard);
    base.GetDetails();
    Console.WriteLine("Contact : " + Contact);
}
```

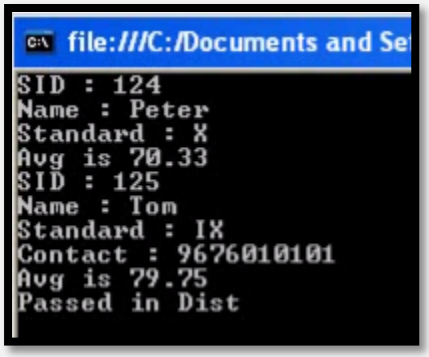
- From here we are calling the GetDetails method of base class.

```
public virtual void GetDetails()
{
    Console.WriteLine("SID : " + Sid);
    Console.WriteLine("Name : " + Name);
    Console.WriteLine("Standard : " + Standard);
}
```

- So it is going to execute this.
- Now it comes back to derived class method.

```
public override void GetDetails()
{
    //Console.WriteLine("SID : " + Sid);
    //Console.WriteLine("Name : " + Name);
    //Console.WriteLine("Standard : " + Standard);
    base.GetDetails();
    Console.WriteLine("Contact : " + Contact);
}
```

- And execute remaining part over here.



- So here we've seen two uses of **base** keyword in the same way we've saw two uses of **this** keyword.
- One use is to invoke the constructor of base class from the derived class.
- And another use is to refer the members of base class as we were using this keyword to refer to the member of current class.
- So here we are utilizing the base keyword and the inheritance concept.
- And the constructor chaining concept and all this things we've implemented in this example.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MethodOverridingEg
{
    class Student
    {
        protected int Sid;
        protected string Name;
        protected string Standard;
```

```
public Student()
{
}

public Student(int Sid, string Name, string Standard)
{
    this.Sid = Sid;
    this.Name = Name;
    this.Standard = Standard;
}

protected double Maths, Science, Social;

public double _Maths
{
    set { Maths = value; }
    get { return Maths; }
}

public double _Science
{
    set { Science = value; }
    get { return Science; }
}

public double _Social
{
    set { Social = value; }
```

```
            get { return Social; }
        }

        public virtual void GetDetails()
        {
            Console.WriteLine("SID : " + Sid);
            Console.WriteLine("Name : " + Name);
            Console.WriteLine("Standard : " + Standard);
        }

        public virtual void GetAvg()
        {
            Console.WriteLine("Avg is " +Math.Round((Maths + Science + Social) / 3.0,2));
        }
    }

    class NewStudent : Student
    {
        protected string Contact;

        public NewStudent(int Sid, string Name, string Standard,string
Contact):base(Sid,Name,Standard)
        {
            //this.Sid = Sid;
            //this.Name = Name;
            //this.Standard = Standard;
            this.Contact = Contact;
        }
```

```
protected double English;

public double _English
{
    set { English = value; }
    get { return English; }
}

public override void GetDetails()
{
    //Console.WriteLine("SID : " + Sid);
    //Console.WriteLine("Name : " + Name);
    //Console.WriteLine("Standard : " + Standard);
    base.GetDetails();
    Console.WriteLine("Contact : " + Contact);
}

public override void GetAvg()
{
    Console.WriteLine("Avg is " + Math.Round((Maths + Science + Social + English) / 4.0, 2));
}

public void GetDivision()
{
    double avg = (Maths + Science + Social + English) / 4.0;
    if (avg < 35)
        Console.WriteLine("Fail");
    else if(avg<60)
        Console.WriteLine("Pass");
```

```
        else if(avg<70)
            Console.WriteLine("Passed in first Division");
        else if(avg<=100)
            Console.WriteLine("Passed in Dist");
    }
}

class Program
{
    static void Main(string [] args)
    {
        Student s = new Student(124, "Peter", "X");
        s._Maths = 67;
        s._Science = 78;
        s._Social = 66;
        s.GetDetails();
        s.GetAvg();


        NewStudent ns = new NewStudent(125, "Tom", "IX", "9676010101");
        ns._English = 77;
        ns._Maths = 89;
        ns._Science = 55;
        ns._Social = 98;

        ns.GetDetails();
        ns.GetAvg();
        ns.GetDivision();
```

```
            Console.ReadLine();
        }
    }
}
```

CHAPTER 17 : ABSTRACT METHOD AND ABSTRACT CLASS IN C#.

- In this chapter we're going to see an abstract method and an abstract class.
- In our earlier chapter we've seen virtual method and we've seen virtual keyword.
- Virtual method is nothing but it is a method which is declared with the keyword virtual.
- And it is up to the derived class whether to override it or not to override it.
- That means it is wish of derived class to override a virtual method.
- If the derived class wants to override the virtual method then can override it.
- If the derived class doesn't wants to override the virtual method then there is no problem.
- It's up to the derived class whether to override the virtual method or not.
- Now here we are going to see an abstract method.
- Abstract method is nothing but it is a method which is declare with the keyword abstract.
- Now let us create a class A.
- In this we've an abstract method show.

```
class A
{
    public abstract void show();
}
```

- So abstract method is a method which is declare with the keyword abstract and it does not have the body.
- We've one non abstract method in class i.e. Display.

```
abstract class A
 {
     public abstract void show();

     public void Display()
     {
         Console.WriteLine("This is Display method");
     }
 }
```

- Now let us create an object of this class.

```
class Program
{
    static void Main(string[] args)
    {
        A a = new A();

        Console.ReadLine();
    }
}
```

- But whenever we try to execute this it throws an error as 'show () is abstract but it is contained in non-abstract class'.
- It means that whenever we have abstract method in the class then the class become abstract.
- So we need to make class as abstract.
- Now let us execute.
- But we got another error.

```
static void Main(string[] args)
{
    A a = new A();
          Cannot create an instance of the abstract class or interface 'AbstractEg.A'
    Console.ReadLine();
}
```

- It says that 'cannot create an instance of the abstract class'.
- It says cannot create the object of an abstract class.

- But we can create the reference.

```
static void Main(string[] args)
{
    A a;

    Console.ReadLine();
}
```

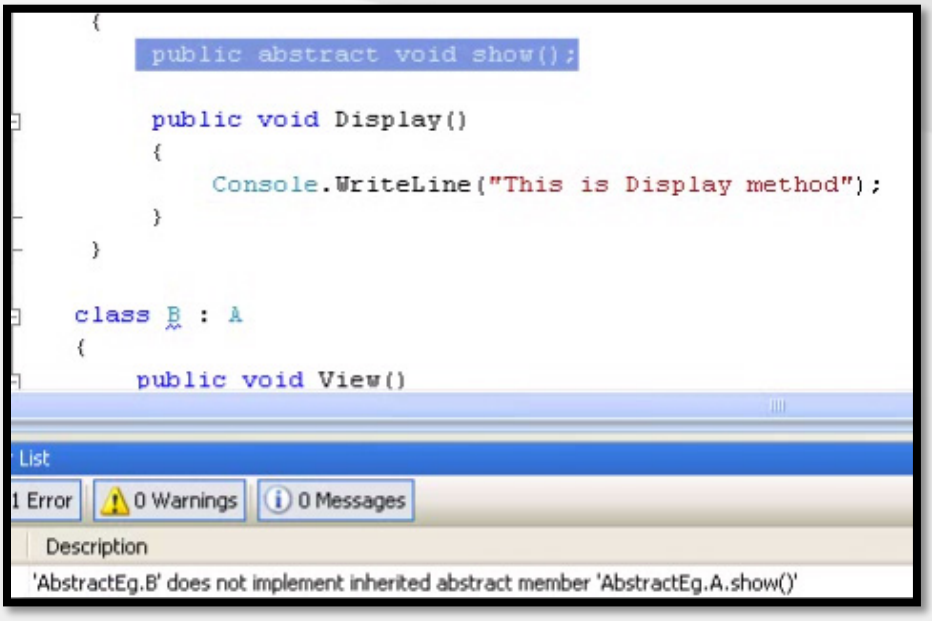
- Here ‘a’ is the reference as we’ve seen in earlier chapter.
- We can create the reference of an abstract class.
- Now it will be execute successfully.
- But we should get a doubt when we cannot create the object then what is the use?
- We’ll definitely see in real time implementation of an abstract method and an abstract class.
- As of now we’ll try to understand what is abstract method and what is abstract class.
- So abstract method is nothing but it is a method which is declare with the keyword abstract and it does not have the body.
- Abstract method has only declaration i.e. prototype.
- And if a class contain at least one abstract method then the class becomes abstract.
- We cannot create the object of an abstract class.
- And we can create only the reference of abstract class.
- For example we’ve one more class B.
- And we are inheriting class A into class B.
- In this we’ll create one method i.e. view.

```
class B : A
{
    public void View()
    {
        Console.WriteLine("This is View Method");

    }
}
```

- Now we can create the object of class B.
- Using the object of class B we can call display and view.
- And we can call show method also because it is public but there is no implementation here.

```
A a;

B b = new B();

b.Display();
b.View();
b.show()
```

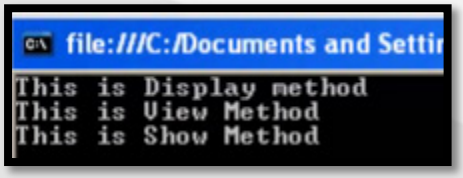
- Let us execute what it says.
- There is an error.



- It means that if we inheriting a class A which is abstract then it is must for us to override the abstract method or to implement the abstract method.
- This is one more point about abstract.
- So we need to implement the abstract method in class B.

```
class B : A
{
    public override void show()
    {
        Console.WriteLine("This is Show Method");
    }
    public void View()
    {
        Console.WriteLine("This is View Method");

    }
}
```

- Now let us execute this.



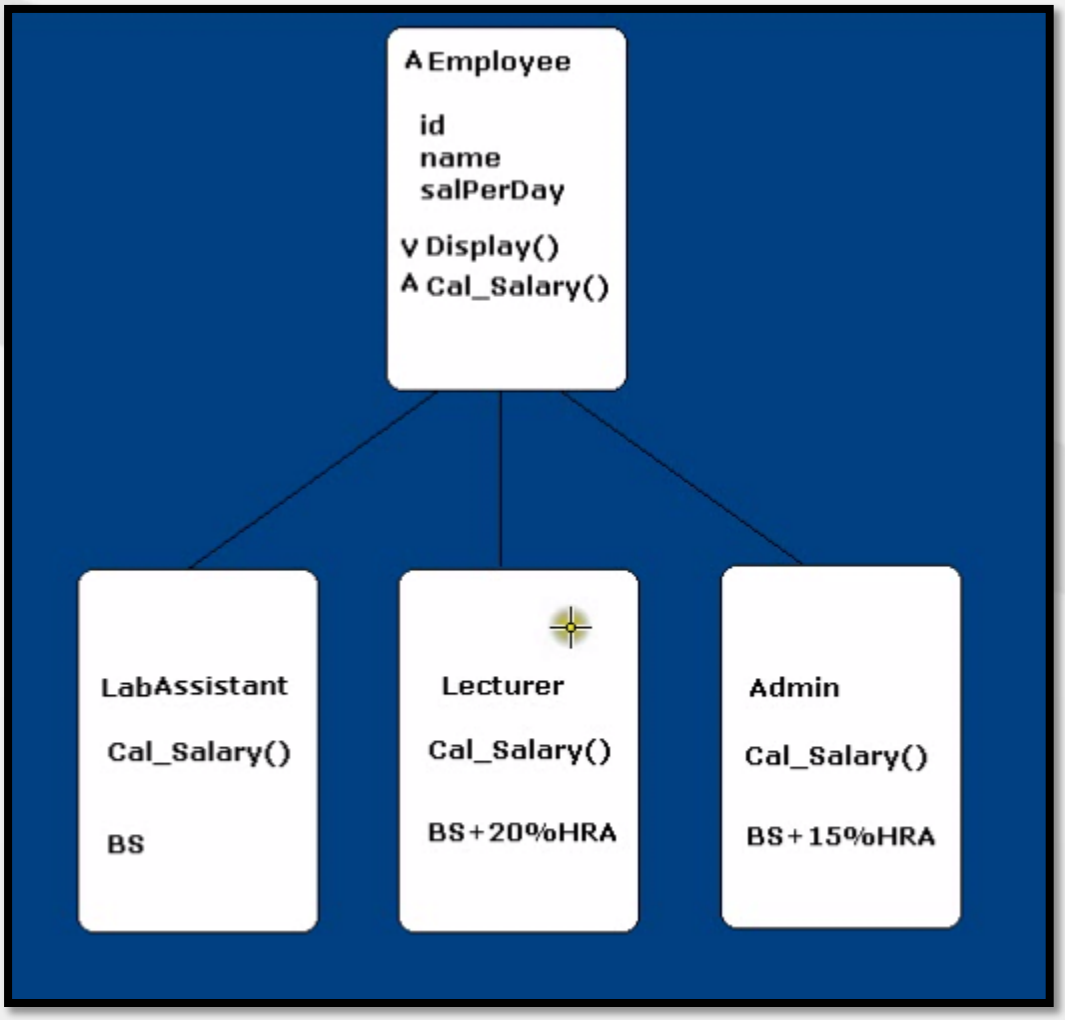
- So this is perfectly showing all the methods.
- Here we've seen abstract method.
- If we compare **abstract** method with **virtual** method.
- In **virtual** method it was up to the derived class whether to override it or not to override it.
- But in **abstract** method it is simply opposite, it becomes must for derived class to override the abstract method.
- So this is all about abstract method and abstract class.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AbstractEg
{
   abstract class A
    {
        public abstract void show();

        public void Display()
        {
            Console.WriteLine("This is Display method");
        }
    }

   class B : A
   {
       public override void show()
       {
           Console.WriteLine("This is Show Method");
       }
       public void View()
       {
           Console.WriteLine("This is View Method");

        }
   }
```

```
    class Program
    {
        static void Main(string [] args)
        {
            A a;

            B b = new B();

            b.Display();
            b.View();
            b.show();

            Console.ReadLine();
        }
    }
}
```

CHAPTER 18 : REAL-TIME RUNTIME POLYMORPHISM IMPLEMENTATION IN C#.

- In this chapter we'll see what is a runtime polymorphism?
- So runtime polymorphism is a big concept and we'll see in details.
- We've seen abstract classes in earlier chapters we saw the complete concept of runtime polymorphism.
- Now we'll go for its implementation.
- We'll understand the runtime polymorphism with the following example.



- So we've a class Employee which is an abstract class which is blue print of the problem.
- Employee has employee id, name and salary per day.
- We're trying to develop an application for a college.
- Lab assistant, Lecturer and Admin are nothing but the roles or the end users of our application.
- And we've employees class which is an abstract class, so lab assistant we're inheriting from employee class.
- Lecturer we're inheriting from employee class.
- Admin we're inheriting from employee class.
- So Lab assistant, Lecturer and Admin are employees.
- We've employee id, name and SalPerDay common for all the end users so we've taken than in our abstract class.
- And we've virtual method Display() to display the details.
- And we've abstract method Cal_Salary().
- Calculate salary for different employees are different so we haven't implemented the calculate salary method just it's a blue print.
- So any class which is inheriting employee they need to implement calculate salary.
- Lab assistant implementing calculate salary for them they have only basic salary.
- That means salary per day * number of days present.
- For lecturer it is BS+20%HRA here implementation is different.
- And for admin it is BS+15%HRA here also implementation is different.
- So we can create the reference of abstract class and at runtime we can ask them to give input for calculate salary for different employees.

- First of all we'll try to implement with two classes employee and lab assistant then we'll go for lecturer and admin.
- Initially we'll make the reference of an employee as an object of lab assistant.
- First of all we'll implement employee class.
- See the implementation.

```
abstract class Employee
{
    protected int EmpId;
    protected string EmpName;
    protected double SalPerDay;

    public Employee(int EmpId, string EmpName, double SalPerDay)
    {
        this.EmpId = EmpId;
        this.EmpName = EmpName;
        this.SalPerDay = SalPerDay;
    }

    public virtual void Display()
    {
        Console.WriteLine("EmpId : " + EmpId);
        Console.WriteLine("EmpName : " + EmpName);
        Console.WriteLine("Emp Sal Per Day : " + SalPerDay);
    }

    public abstract void CalculateSalary(int days);
}
```

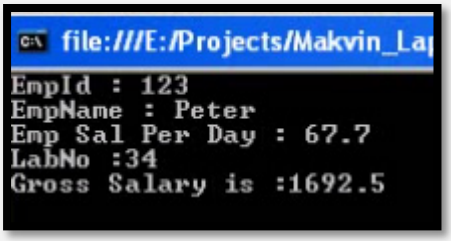
- Here we've implemented an abstract class employee with constructor, display as virtual method and calculate salary as abstract method.
- In that method we've pass one parameter for number of days present.
- And our class is abstract.
- Now we'll inherit in lab assistant class.
- Now we'll create new class lab assistant which inherits Employee class.

```
class LabAssistant : Employee
{
    protected int LabNo;
    public LabAssistant(int EmpId, string EmpName, double SalPerDay,int LabNo):base(EmpId,EmpName,SalPerDay)
    {
        this.LabNo = LabNo;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine("LabNo :" + LabNo);
    }

    public override void CalculateSalary(int days)
    {
        double GS = SalPerDay * days;
        Console.WriteLine("Gross Salary is :"+GS);
    }
}
```

- Here we've created a new class Lab Assistant which inherits employee class.
- We've used base keyword to initialize some fields at base class only.
- And remaining field i.e. lab number is initializing here itself.
- And we've Display() as override method.
- In that we've used base keyword for displaying some fields in base method only and one lab number is displaying here.
- And we've implemented abstract method.
- Now we'll create a reference of an employee class.
- And we make it an object of lab assistant.
- And we'll call Display and calculate salary methods as below.

```
Employee E;
E = new LabAssistant(123, "Peter", 67.7, 34);
E.Display();
E.CalculateSalary(25);
```

- Now press F5.



- Here it is displaying employee details.
- And it's also displaying Gross Salary i.e. only basic salary= number of days present * salary per day.
- This is fine for one class.
- Let it be the by default object of lab assistant.
- In the same way we'll create remaining two classes that means for lecturer and admin.
- Now we'll create the class for lecturer.
- Lecturer will not have lab number, lecturer has subject.

```
class Lecturer : Employee
{
    protected string Sub;
    public Lecturer(int EmpId, string EmpName, double SalPerDay, string Sub)
        : base(EmpId, EmpName, SalPerDay)
    {
        this.Sub = Sub;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine("Sub :" + Sub);
    }

    public override void CalculateSalary(int days)
    {
        double GS = (SalPerDay * days)+((SalPerDay * days)*20/100);
        Console.WriteLine("Gross Salary is :" + GS);
    }
}
```

- Here we've override method calculate salary which is displaying Gross Salary which includes 20% of HRA.
- In the same way we'll create admin class.
- Admin class we do not have any extra fields.
- So simply we'll call base class constructor from derived class.
- And for calculate salary we'll change it as 15% for HRA.

```
class Admin : Employee
{
    public Admin(int EmpId, string EmpName, double SalPerDay) : base(EmpId, EmpName, SalPerDay)
    {

    }

    public override void CalculateSalary(int days)
    {
        double GS = (SalPerDay * days) + ((SalPerDay * days) * 15 / 100);
        Console.WriteLine("Gross Salary is :" + GS);
    }
}
```

- Now we'll give the choices for end user to select.
- And we'll read this choice in variable 'ch'.

```
Console.WriteLine("Enter You Choice");
Console.WriteLine("1.LabAssistant 2.Lecturer 3.Admin 4.Exit");
int ch = int.Parse(Console.ReadLine());
```

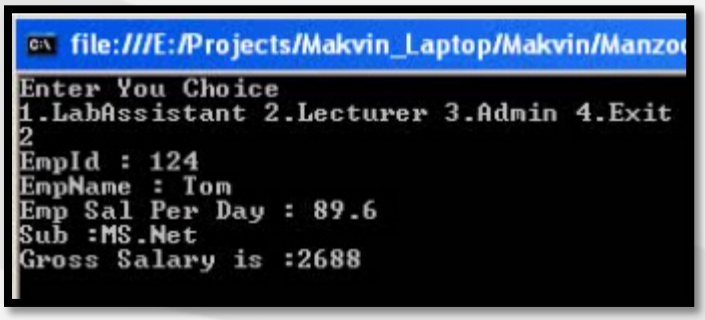
- Depending upon the choices we'll make the reference the object of that particular class.
- By default it will be an object of lab assistant.
- So we'll use switch to that choice 'ch'.
- In switch we'll give cases for Lab assistant, lecturer and admin.

```
Employee E;
E = new LabAssistant(123, "Peter", 67.7, 34);

Console.WriteLine("Enter You Choice");
Console.WriteLine("1.LabAssistant 2.Lecturer 3.Admin 4.Exit");
int ch = int.Parse(Console.ReadLine());

switch (ch)
{
    case 1:
        E = new LabAssistant(123, "Peter", 67.7, 34);
        break;

    case 2:
        E = new Lecturer(124, "Tom", 89.6, "MS.Net");
        break;

    case 3:
        E = new Admin(126, "Lilly", 45.8);
        break;
```

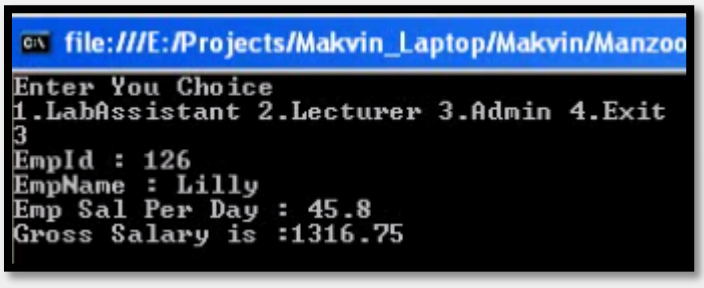
```
        case 4:
            System.Threading.Thread.CurrentThread.Abort();
            break;

        default:
            break;
    }

    E.Display();
    E.CalculateSalary(25);

    Console.ReadLine();
```

- If we observe this program can we say that what is the method it is going to display or whose salary it is going to calculate at compile time?
- When we execute the end user may give option 1 so it will display methods from lab assistant and calculate the salary of lab assistant.
- If the end user give option 2 so it will displays methods from lecturer and calculate the salary of lecturer.
- If the end user give option 3 so it will displays methods from admin and calculate the salary of admin.

- If the case is 4 then it is going to ends the program.
- So at compile time we can't say which class method is going to execute.
- So we cannot say that at compile time why because user may give any option.
- So depending upon that option it is going to generate method for that.
- So the concept behind this is 'runtime polymorphism'.
- We need to call display and calculate salary method only once and depending upon the user choice we are making the object at runtime.
- We're making the reference of base class and making it an object of derived class at runtime and then we are calling display and calculate salary methods
- Let us execute this.

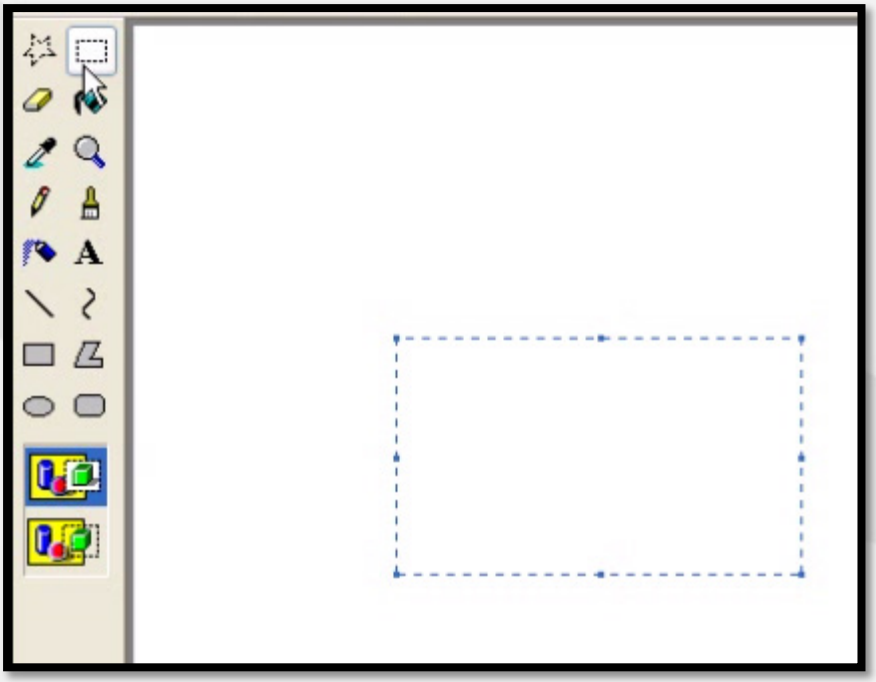


- Our option is 2.
- So it's going to calculate the salary for lecturer.
- We'll execute this again.

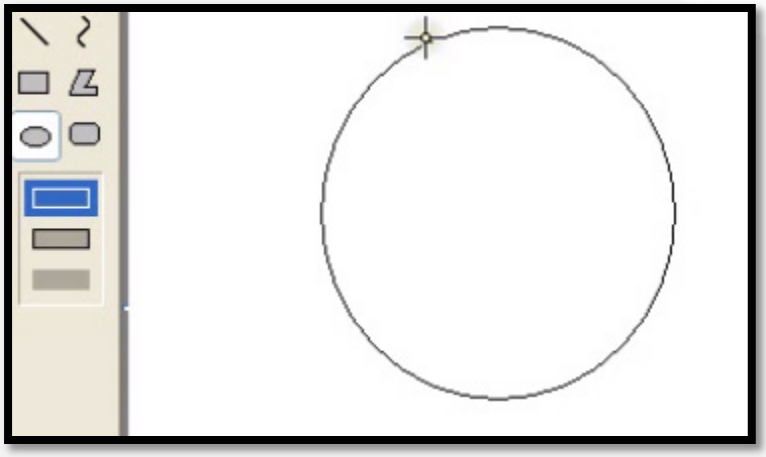


- Now our option is 3.

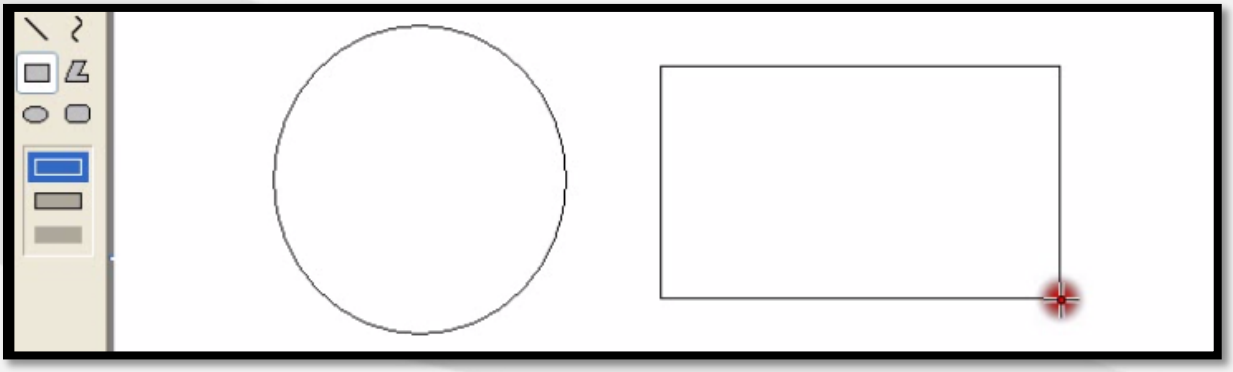
- So it's going to calculate the salary for admin.
- So at runtime its calculating the salary
- If our option is 4 it's going to stop the program.
- So we're making the reference of abstract class and making it an object of derived class at runtime.
- And depending upon the user input it's going to calculate the salary of lab assistant or lecturer or admin.
- So this concept is called as 'runtime polymorphism'.
- If we observe our paint also works on runtime polymorphism.
- In this paint we've the class reference of class diagram.



- If we click drag and drop it is going to select the area because our reference become select area.
- Now we'll make it an object of circle and we'll execute same method click drag and drop and leave it will draw circle.



- Because now our object reference is circle.
- If we select on rectangle now our object reference become rectangle.
- And performing same method click drag drop and leave.



- Now it's drawn rectangle.
- That means at runtime it is deciding to make it an object of what class.
- If we select line it becomes an object of line class.
- If we select on magnifier it becomes an object of magnifier class.
- So our paint is working on runtime polymorphism.
- So we see that we are calling single method but its calling various task depending upon the object that it's make at runtime.
- So this is nothing but **Runtime Polymorphism** real-time example.

- We'll have runtime polymorphism concept in almost all big projects.
- So this is all about runtime polymorphism.
- Let us define runtime polymorphism.
- Runtime polymorphism is nothing but if we have an abstract class and n number of classes which is implementing the abstract class and we're making the reference of abstract class and making it an object of any class at runtime and calling the methods.
- The methods code gets generated at runtime depending upon the users input.
- So this complete concept is called as runtime polymorphism.
- Runtime in the sense at the time of execution.
- Polymorphism means many meanings.
- Calculate salary has many meanings.
- It is calculate the salary of admin, lectures and lab assistant but depends at runtime.
- At compile time we cannot say what method it is going to execute.
- So this is all about runtime polymorphism.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RuntimePolymorphismEg
{
    abstract class Employee
```

```
{
    protected int EmpId;
    protected string EmpName;
    protected double SalPerDay;

    public Employee(int EmpId, string EmpName, double SalPerDay)
    {
        this.EmpId = EmpId;
        this.EmpName = EmpName;
        this.SalPerDay = SalPerDay;
    }

    public virtual void Display()
    {
        Console.WriteLine("EmpId : " + EmpId);
        Console.WriteLine("EmpName : " + EmpName);
        Console.WriteLine("Emp Sal Per Day : " + SalPerDay);
    }

    public abstract void CalculateSalary(int days);
}

class LabAssistant : Employee
{
    protected int LabNo;
    public LabAssistant(int EmpId, string EmpName, double SalPerDay,int
LabNo):base(EmpId,EmpName,SalPerDay)
    {
        this.LabNo = LabNo;
```

```
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine("LabNo :" + LabNo);
    }

    public override void CalculateSalary(int days)
    {
        double GS = SalPerDay * days;
        Console.WriteLine("Gross Salary is :"+GS);
    }
}

class Lecturer : Employee
{
    protected string Sub;
    public Lecturer(int EmpId, string EmpName, double SalPerDay, string Sub)
        : base(EmpId, EmpName, SalPerDay)
    {
        this.Sub = Sub;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine("Sub :" + Sub);
    }
```

```
    public override void CalculateSalary(int days)
    {
        double GS = (SalPerDay * days)+((SalPerDay * days)*20/100);
        Console.WriteLine("Gross Salary is :" + GS);
    }
}

class Admin : Employee
{

    public Admin(int EmpId, string EmpName, double SalPerDay) : base(EmpId, EmpName, SalPerDay)
    {

    }

    public override void CalculateSalary(int days)
    {
        double GS = (SalPerDay * days) + ((SalPerDay * days) * 15 / 100);
        Console.WriteLine("Gross Salary is :" + GS);
    }
}

class Program
{
    static void Main(string [] args)
    {
        Employee E;
```

```
E = new LabAssistant(123, "Peter", 67.7, 34);

Console.WriteLine("Enter You Choice");
Console.WriteLine("1.LabAssistant 2.Lecturer 3.Admin 4.Exit");
int ch = int.Parse(Console.ReadLine());

switch (ch)
{
    case 1:
        E = new LabAssistant(123, "Peter", 67.7, 34);
        break;

    case 2:
        E = new Lecturer(124, "Tom", 89.6, "MS.Net");
        break;

    case 3:
        E = new Admin(126, "Lilly", 45.8);
        break;

    case 4:
        System.Threading.Thread.CurrentThread.Abort();
        break;

    default:
        break;
}

E.Display();
```

```
            E.CalculateSalary(25);


            Console.ReadLine();
        }
    }
}
```

CHAPTER 19 : SEALED METHOD AND SEALED CLASS IN C#.

- In this chapter we'll see what a sealed method is and what is a sealed class.
- Now we'll use the code of abstract class.

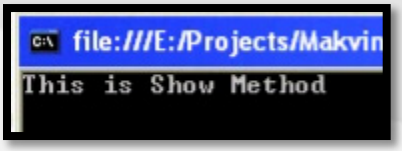
```
abstract class A
{
    public abstract void show();

    public void Display()
    {
        Console.WriteLine("This is Display method");
    }
}

class B : A
{
    public override void show()
    {
        Console.WriteLine("This is Show Method");
    }
    public void View()
    {
        Console.WriteLine("This is View Method");

    }
}
```

- If we observe we've an abstract method if we want to override that abstract method we can override it in derived class.
- Now because we've implemented the abstract method in class B
- So class B is not an abstract class anymore, now it is not compulsory for us to override show() method again.
- We'll go for multi-level inheritance and create the object of class C and call show method.
- It is going to execute its base class method.

```
class B : A
{
    public override void show()
    {
        Console.WriteLine("This is Show Method");
    }
    public void View()
    {
        Console.WriteLine("This is View Method");

    }
}

class C : B
{

}
```

```
class Program
{
    static void Main(string[] args)
    {
        C c = new C();
        c.show();

        Console.ReadLine();
    }
}
```

- Press F5.



- Base class for C is class B so it is going to execute class B show method.
- If we want to override this method definitely we can override it.

```
class C : B
{
    public override void show()
    {
        Console.WriteLine("This is Show of C");
    }
}
```

- Now it is going to execute this new method.

- This concept we've already seen.
- Now what we want, we want to stop the method overriding at class B itself.
- Means we don't want the derived class to override that show method again.
- Like first case it was up to the derived class to override the method so we were using virtual keyword.
- In second case it is must for the derived class to override the method so we were using abstract keyword.
- Now we want restrict the method overriding that means we don't want the derived class to override the method.
- If derived class tries to override the method then we'll stop that.
- We can do that with the help of keyword **sealed**.
- If we try to override it throws an error.
- If a method is sealed we cannot override it.
- But do remember that it is not like abstract keyword if method is abstract then class become abstract.
- In a normal class we can have sealed method.
- But do remember that sealed keyword is always used with the combination of override keyword.
- That means we cannot override a sealed method.
- And the class which is inheriting a class with sealed method cannot override the sealed method.
- If a class is sealed we cannot extend or we cannot derived new class from sealed class.

```
sealed class B : A
 {
     public override sealed void show()
     {
         Console.WriteLine("This is Show Method");
     }
     public void View()
     {
         Console.WriteLine("This is View Method");

     }
 }

 class C : B
 {
     //public override void show()
     //{
     //    Console.WriteLine("This is Show of C");
     //}
 }
```

- So it's shows an error "we cannot derived new class from sealed class".
- But we can create an object of sealed class.

```
B b = new B();

Console.ReadLine();
```

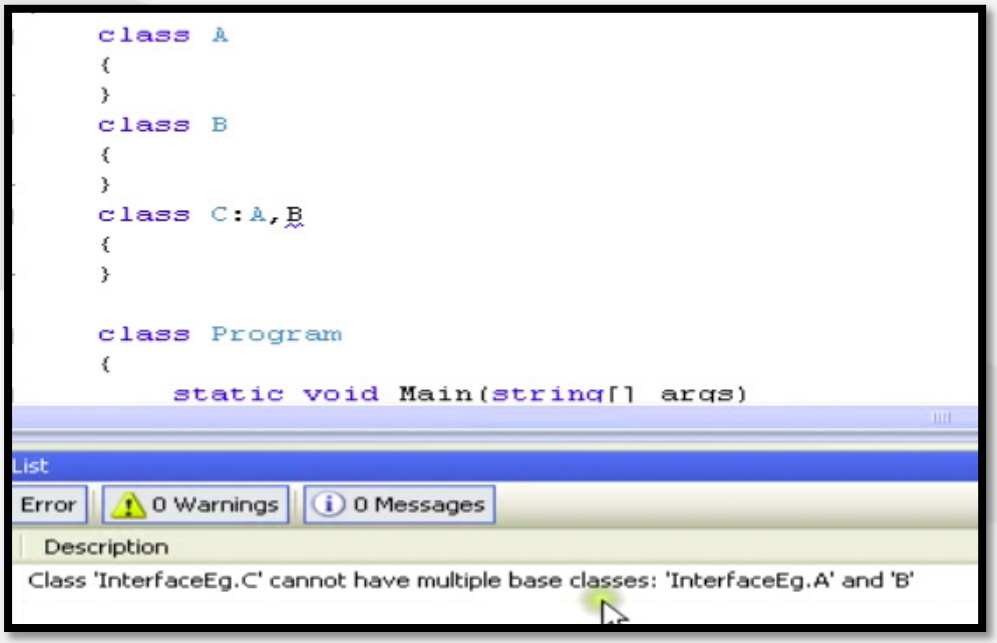
- It is very simple.
- And it's not like abstract if method is abstract class must be abstract.
- Here if method is sealed there may be class is sealed or may not be sealed.
- One more difference is we cannot create an object of abstract class.
- Whereas in sealed we can create an object of sealed class.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SealedEg
{
    abstract class A
    {
        public abstract void show();

        public void Display()
        {
            Console.WriteLine("This is Display method");
        }
    }

   sealed class B : A
    {
        public override sealed void show()
        {
            Console.WriteLine("This is Show Method");
        }
        public void View()
        {
            Console.WriteLine("This is View Method");

        }
    }
```

```
    //class C : B
    //{
    //    //public override void show()
    //    //{
    //    //    Console.WriteLine("This is Show of C");
    //    //}
    //}

    class Program
    {
        static void Main(string [] args)
        {


            B b = new B();


            Console.ReadLine();
        }
    }
}
```

CHAPTER 20 : INTERFACES AND MULTIPLE INHERITANCE IN C#.

- In this chapter we'll see what is an interface and what is the use of an interface.
- So till now we've seen inheritance in classes and we implemented single level inheritance and multi-level inheritance.
- Now if we've a situation where we want to implement multiple inheritance.
- That means if we have two base classes and single derived class then it's known as multiple inheritance but this kind of inheritance is not possible using classes.



- See it throws an error.
- We cannot have multiple base classes.
- There must be one and only one base class.
- But if we've a situation where we want to implement multiple inheritance then we can do that with the help of **Interfaces.**

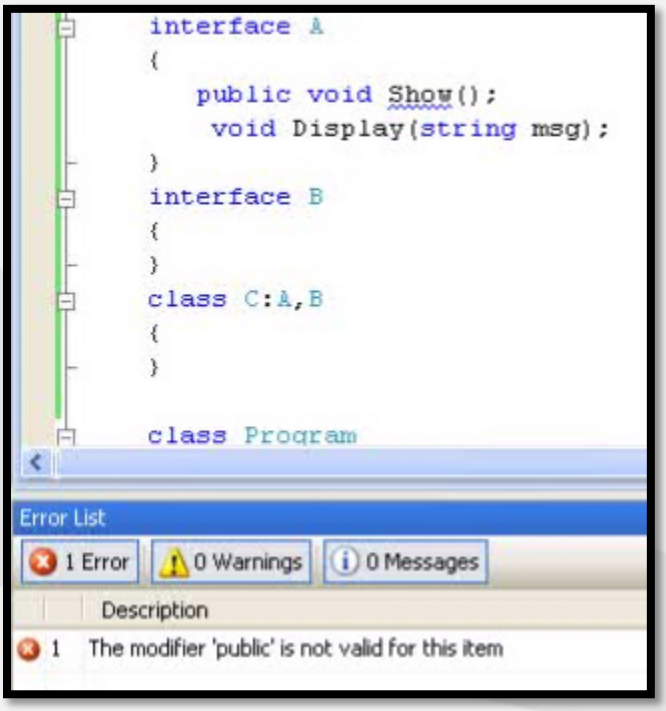
- We can declare two interfaces then we can go for multiple inheritance.
- See code below.

```
interface A
{
}
interface B
{
}
class C:A,B
{
}
```

- Whenever we have a scenario where we want to implement multiple inheritance then we can go for inheritance.
- It is also the blue print of the problem.
- And we can say pure abstract class.
- That means it will have abstract members.
- It defines what is that we should do.
- Interface is also the blue print of the problem.
- An Interface contains methods we cannot have fields, constructors, destructors or static members inside an interface.
- An interface can contains methods, properties and events.
- It is pure abstract class.
- We need not say the methods inside an interface is abstract using abstract keyword explicitly.
- We can simply give the prototype.

```
interface A
{
    void Show();
    void Display(string msg);
}
```

- These are the prototypes of two methods.
- Abstract methods are methods which doesn't have body and declare with the keyword abstract.
- Whereas in interface if we declare the methods by default it becomes abstract as well as public.
- We need not to give the methods as public and abstract explicitly.
- If we give public it will not allow us.



- So we cannot give public explicitly.
- By default they are public and abstract.
- If we inheriting both the interfaces then it becomes must for us to implement all the members of both interfaces A and B.
- If we run the program without implementation then it throws an error.

```
interface A
{
    void Show();
    void Display(string msg);
}
interface B
{
    void View();
}
class C:A,B
{
}
```

Error List

3 Errors | 0 Warnings | 0 Messages

	Description
1	'InterfaceEg.C' does not implement interface member 'InterfaceEg.B.View()'
2	'InterfaceEg.C' does not implement interface member 'InterfaceEg.A.Display(string)'
3	'InterfaceEg.C' does not implement interface member 'InterfaceEg.A.Show()'

- Here it displays error as derived class does not implement interface members.
- Means we've to implement interface members must.
- At the time of implementation we need to use as public keyword.

```
class C:A,B
{
    public void Show()
    {
        Console.WriteLine("This is Show Method");
    }

    public void Display(string msg)
    {
        Console.WriteLine("This is Display of "+msg);
    }
    public void View()
    {
        Console.WriteLine("This is View Method");
    }
}
```

- Now we've implemented all the abstract members.
- Now we'll create the object of c.
- And we'll call all the methods.

```
C c = new C();
c.Display("ManzoorTheTrainer");
c.Show();
c.View();
Console.ReadLine();
```

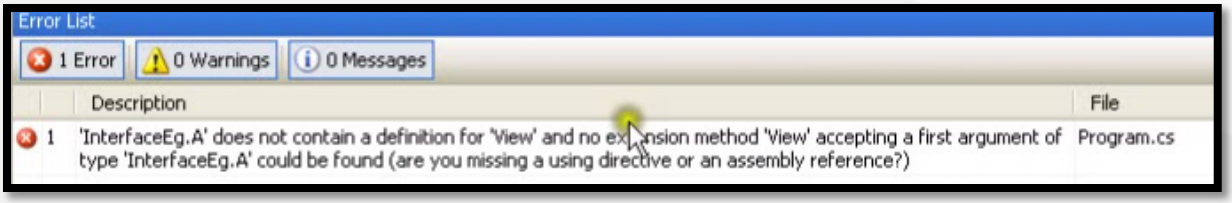
- Press F5.

```
file:///C:/Documents and Settings/makvin/Loc
This is Display of ManzoorTheTrainer
This is Show Method
This is View Method
```

- Now it works fine.
- Its pure abstract class.
- And we can implement runtime polymorphism using interfaces also.
- So let's go for implementation.
- Now we'll create the reference of interface A.
- And call all the methods.

```
A a;

a = new C();
a.Display("ManzoorTheTrainer");
a.Show();
a.View();
```

- Press F5.


Error List
1 Error
0 Warnings
0 Messages
Description
File
1 'InterfaceEg.A' does not contain a definition for 'View' and no extension method 'View' accepting a first argument of type 'InterfaceEg.A' could be found (are you missing a using directive or an assembly reference?)
Program.cs

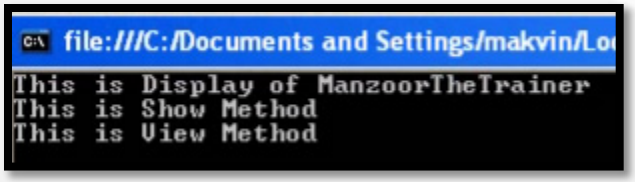
- But it throws an error.
- It says does not contain definition of view.
- Because we are making reference of interface A.
- But we do not have view prototype in interface A.

```
interface A
{
    void Show();
    void Display(string msg);
}
```

- That's way we cannot invoke view method using the reference of interface A.
- If we making reference of interface A then we can invoke those methods which are available in interface A.
- So we need to create the reference of interface B to invoke view method.

```
B b;
b = new C();
b.View();
```

- And we cannot invoke neither display nor show methods using reference of B.
- Because interface B has only view method definition.
- If we try to access then it shows an error.
- Now if we execute.



- Now its work fine.
- So using the reference of interface A we cannot invoke the methods of interface B.
- And using the reference of interface B we cannot invoke the methods which are defined in interface A.
- So it's same as abstract classes and we can implement runtime polymorphism.
- Because we are creating reference of an interface and we can make an object at runtime depending upon user input.
- So we can implement runtime polymorphism using interfaces.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceEg
{
    interface A
    {
        void Show();
        void Display(string msg);
    }
    interface B
    {
```

```
    void View();
}
class C:A,B
{
    public void Show()
    {
        Console.WriteLine("This is Show Method");
    }

    public void Display(string msg)
    {
        Console.WriteLine("This is Display of  "+msg);
    }
    public void View()
    {
        Console.WriteLine("This is View Method");
    }
}

class Program
{
    static void Main(string [] args)
    {
        //C c = new C();
        //c.Display("ManzoorTheTrainer");
        //c.Show();
        //c.View();
        A a;
```

```
            a = new C();
            a.Display("ManzoorTheTrainer");
            a.Show();
            //a.View();

            B b;
            b = new C();
            b.View();


            Console.ReadLine();
        }
    }
}
```

CHAPTER 21 : COLLECTION CLASSES AND ITS LIMITATION IN C#.

- In this chapter we'll see collection classes that we have.
- All our collection classes present in the namespace called as System.Collections.
- Using **System. Collections**;
- Here we are going to demonstrate two collection classes.
- One is stack, and another is Queue.
- If we want to store some elements or objects in stack we need not to write all the logic that we've written in earlier C programming.
- But we've readymade classes stack and queue.

```
static void Main(string[] args)
{
    Stack S = new Stack();
}
```

- Here we've created an object for stack.
- Now we can add the elements in stack using method 'push'.

```
Stack S = new Stack();
S.Push(12);
S.Push(14);
S.Push(56);
S.Push(34);
```

- Now we've pushed 4 elements into stack.
- And our stack works like LIFO (Last in first out).
- If we want to pop or remove one element from stack then it removes 34 because we've stored 34 at last.

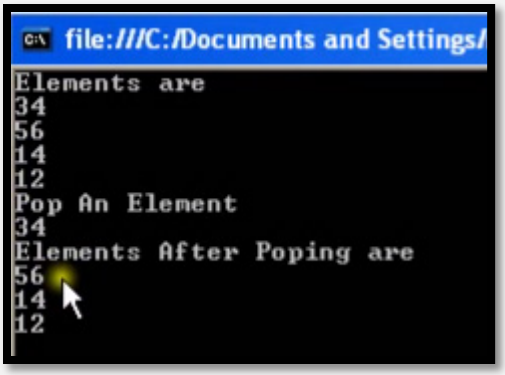
```
Console.WriteLine(S.Pop().ToString());
Console.ReadLine();
```

- Now it'll going to display 34 and removes it.

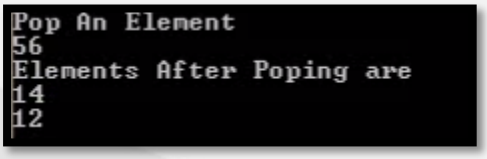
- See it displays 34.
- We can iterate through all the elements using foreach loop.

```
Console.WriteLine("Elements are");
foreach (int item in S)
{
    Console.WriteLine(item);
}

Console.WriteLine("Pop An Element");
Console.WriteLine(S.Pop().ToString());

Console.WriteLine("Elements After Poping are");
foreach (int item in S)
{
    Console.WriteLine(item);
}
```

- Here we're first displaying all the stack elements.
- Next we're poping an element i.e. 34.
- Finally we're displaying remaining elements of stack i.e.56, 14, and 12.
- See below.



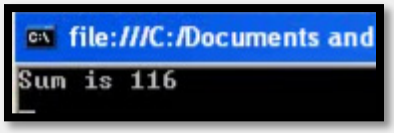
- It works fine as expected.
- If we perform pop operation again then it removes 56 from the stack because it is the last element that we've stored.
- And it'll displays only two elements i.e.14 and 12.



- Now we'll add all the elements of stack.
- We need to comment out all above things and write below logic.

```
int sum = 0;
foreach (int item in S)
{
    sum = sum + item;
}
Console.WriteLine("Sum is "+sum);
Console.ReadLine();
```

- Now press F5.



- See its displaying sum of 4 elements.
- It's fine.

- As like stack we've another class **Queue**.
- Queue has **Enqueue** and **Dequeue** methods for insert and delete operations.
- Queue works like FIFO (first in first out) manner.
- See the implementation.

```
Queue Q = new Queue();
```

- We've declared Queue.
- Now we'll add some elements.

```
Q.Enqueue(23);
Q.Enqueue(45);
Q.Enqueue(56);
Q.Enqueue(458);
```

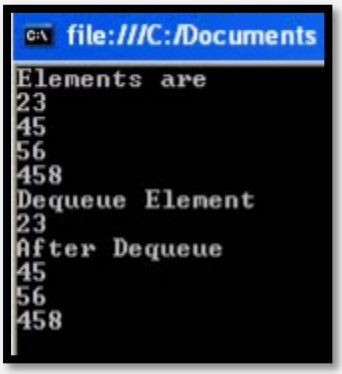
- We've inserted 4 elements.
- Now we can iterate through all elements using foreach loop.

```
Console.WriteLine("Elements are");
foreach (int item in Q)
{
    Console.WriteLine(item);
}
```

- Now we'll perform Dequeue operation and display elements after removing.

```
Console.WriteLine("Elements are");
foreach (int item in Q)
{
    Console.WriteLine(item);
}

Console.WriteLine("Dequeue Element");
Console.WriteLine(Q.Dequeue().ToString());

Console.WriteLine("After Dequeue");
foreach (int item in Q)
{
    Console.WriteLine(item);
}
```

- Now press F5.



- Here it's showing first 4 elements.
- After dequeue an element i.e. 23.
- It's showing 3 elements.
- Here it removes 23 because we've inserted 23 at first.
- And queue is always works like FIFO manner.
- Here also we can perform same operation of addition on queue elements.

```
int sum = 0;
foreach (int item in Q)
{
    sum = sum + item;
}
Console.WriteLine("Sum is " + sum);
```

- Press F5.

- Here it's showing 582 as sum of queue elements.
- If we observe following code.

```
Queue Q = new Queue();

Q.Enqueue(23);
Q.Enqueue(45);
Q.Enqueue(56);
Q.Enqueue(458);
Q.Enqueue("Manzoor");
```

- We've inserted 4 integers and one string but our compiler will not stop us to doing so because we are passing objects.
- We can Enqueue few strings few double and few integers.
- But the problem will arise at the time of addition.

```
foreach (int item in Q)
{
    sum = sum + item;
}
```

- Because it is thinking every item in collection is integer and it is trying to add it.
- But it will throws a runtime error or exception.
- At the time of compilation it will not throw any kind of error.
- It is going to throw an error at the time of executing this.
- At the occurrence of string it throws an exception due to conversion error from string to int.
- This is one of the drawback of collection classes.
- Because at the time of creating an object we are not specifying what type of object using for queue or stack.
- We can stack integers, doubles and strings.
- And we can also stack some objects of user defined classes.
- So there is no specific type at the time of defining stack or queue.
- So the problem with this is it is not type safe.

- So the solution for this.

- Use of Generic Collection classes.
- This we'll see in next chapter.
- Complete code for collection classes is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace CollectionClassesEg
{
    class Program
    {
        static void Main(string [] args)
        {
            Stack S = new Stack();

            S.Push(12);
            S.Push(14);
            S.Push(56);
            S.Push(34);

            //Console.WriteLine( "Elements are");
            //foreach (int item in S)
```

```
//{
//    Console.WriteLine(item);
//}

//Console.WriteLine("Pop An Element");
//Console.WriteLine(S.Pop().ToString());

//Console.WriteLine("Elements After Poping are");
//foreach (int item in S)
//{
//    Console.WriteLine(item);
//}

//Console.WriteLine("Pop An Element");
//Console.WriteLine(S.Pop().ToString());

//Console.WriteLine("Elements After Poping are");
//foreach (int item in S)
//{
//    Console.WriteLine(item);
//}

//int sum = 0;
//foreach (int item in S)
//{
//    sum = sum + item;
//}
//Console.WriteLine("Sum is "+sum);
```

```
Queue Q = new Queue();

Q.Enqueue(23);
Q.Enqueue(45);
Q.Enqueue(56);
Q.Enqueue(458);
Q.Enqueue("Manzoor");

//Console.WriteLine("Elements are");
//foreach (int item in Q)
//{
//    Console.WriteLine(item);
//}

//Console.WriteLine("Dequeue Element");
//Console.WriteLine(Q.Dequeue().ToString());

//Console.WriteLine("After Dequeue");
//foreach (int item in Q)
//{
//    Console.WriteLine(item);
//}

int sum = 0;
foreach (int item in Q)
{
    sum = sum + item;
}
```

```
            Console.WriteLine("Sum is " + sum);


            Console.ReadLine();
        }
    }
}
```

CHAPTER 22 : GENERIC COLLECTION CLASSES IN C#.

- In this chapter we'll see generic collection classes.
- Now we've a namespace.

```
using System;
using System.Collections.Generic;
```

- Now if we want to store the elements in stack of type integers, so we can create the object of a generic class.

```
Stack<int> S = new Stack<int>();
```

- Now we can push only integers.

```
S.Push(23);
S.Push(89);
```

- If we try to push string then it throws compile time error.

```
S.Push("Manzoor");
```

```
class System.String
Represents text as a series of Unicode characters.

Error:
   The best overloaded method match for 'System.Collections.Generic.Stack<int>.Push(int)' has some invalid arguments
```

- That means we cannot pass string to this.
- If we pass double even then it throws an error.

```
S.Push(34.8);
```

```
struct System.Double
Represents a double-precision floating-point number.

Error:
   The best overloaded method match for 'System.Collections.Generic.Stack<int>.Push(int)' has some invalid arguments
```

- We can pass only integers.

- And rest of the things are same.
- We can pop the element.

```
Console.WriteLine(S.Pop());
```

- It will pop top most element i.e. 34 and it should display.

- See that it has pop 34.
- This is our generic collection class.
- In the same way we can also create queues.

```
Console.WriteLine(S.Pop());

Queue<string> Q = new Queue<string>();
```

- Here we can store only strings.
- This will create a queue where we can store only strings.
- Generic collection classes nothing but strongly typed collection classes.
- We've one more class called as List that would be using very much in our layered architecture or in real time implementation.

```
List<int> L = new List<int>();
```

- Here we can store the elements in the form of integers.
- Like we've arrays here we've lists.
- Best implementation of list could be whenever we've situation of storing and retrieving all the students.
- We can store an objects in the lists.

- For example we've a class called student.
- That contain two fields, one constructor to initialize.
- And one display method to show student data.
- See below.

```
class Student
{
    int Sid;
    string Name;

    public Student(int Sid, string Name)
    {
        this.Sid = Sid;
        this.Name = Name;
    }

    public void Display()
    {
        Console.WriteLine("SID:{0} SName:{1}",Sid,Name);
    }
}
```

- Now we'll create a List of type student.
- Like we are creating list of integers, double and string.
- Here we are creating a List of type student.
- See below.

```
List<Student> S = new List<Student>();
```

- Here we've created List for students.
- We can define the size of list as well.
- By default it will take the size of list as 4.
- If we want to define some size we can do that.

```
List<Student> S = new List<Student>(5);
```

- Here we define the size as 5.
- We can add objects into list using the method as **s.add();.**
- First of all we need to create an object for student, then we can add those objects to the list.

```
List<Student> S = new List<Student>();

Student s1=new Student(124,"Peter");
S.Add(s1);
```

- Here we've create an object s1 for class student.
- Then we've added s1 to the list of students.
- Like that we'll add one more object of student to the list.

```
Student s2 = new Student(125, "Jack");
S.Add(s2);
```

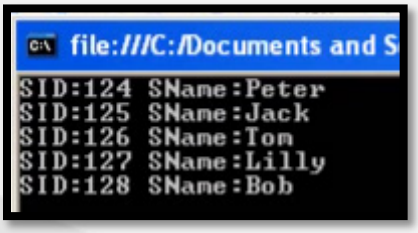
- We've added to records.
- Instead of every time creating the reference then creating an object then adding that reference to the list, we'll add objects of student directly to the list as below.

```
S.Add(new Student(126,"Tom"));
S.Add(new Student(127, "Lilly"));

S.Add(new Student(128, "Bob"));
```

- Once the list is full it doubles its size automatically.
- All this things are happening automatically we need not worry about memory allocation.
- We can iterate through the list using foreach loop as.

```
foreach (Student item in S)
{
    item.Display();
}
```

- This time our type is student and our collection is S.
- And item.Display() method used to display each student object.
- Every time it takes the each object from student and stores in item.
- And displays that item.
- Like that it iterates through all the objects of student.



- Here we've all the records of student.
- Like that we can implement for list of departments or anything.
- List is very important class we are going to work with this a lot.
- So this is all about our generic collection classes in which lists are very important.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace GenericCollectionClassesEg
{
```

```
class Student
{
    int Sid;
    string Name;

    public Student(int Sid, string Name)
    {
        this.Sid = Sid;
        this.Name = Name;
    }

    public void Display()
    {
        Console.WriteLine( "SID:{0} SName:{1}",Sid,Name);
    }
}

class Program
{
    static void Main(string [] args)
    {


        //Stack<int> S = new Stack<int>();

        //S.Push(23);
        //S.Push(89);
        //S.Push(34);
```

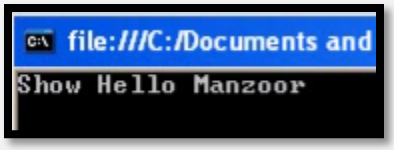
```
            //Console.WriteLine(S.Pop());

            //Queue<string> Q = new Queue<string>();

            //List<int> L = new List<int>();


            List<Student> S = new List<Student>();

            Student s1=new Student(124,"Peter");
            S.Add(s1);

            Student s2 = new Student(125, "Jack");
            S.Add(s2);

            S.Add(new Student(126,"Tom"));
            S.Add(new Student(127, "Lilly"));
            S.Add(new Student(128, "Bob"));


            foreach (Student item in S)
            {
                item.Display();
            }

            Console.ReadLine();
        }
    }
}
```

CHAPTER 23 : CONCEPT OF DELEGATES, UNICAST DELEGATES AND MULTICAST DELEGATES IN C#.

- In this chapter we'll see delegates.
- What are delegates?
- Delegates are nothing but pointers to the functions or methods.
- Now we'll see what do you mean by pointer to the function or pointer to the method.
- We'll create one static method with single parameter.
- If we want to invoke this static method from our main method we can do that because our main method also static.
- From static methods we can access only static members.
- Now we'll just call that method.

```
class Program
{

    static void Main(string[] args)
    {

        Show("Manzoor");
        Console.ReadLine();
    }

    static void Show(string name)
    {
        Console.WriteLine("Show Hello "+name);
    }
}
```

- So this is our simple program.
- Press F5.



- This is our simple show method calling.
- This kind of binding is called as **static Binding**.
- Means code for this method is generated at compile time itself.
- Now we'll call the same method using delegate concept.
- Now we'll create a delegate.

```
class Program
{
    delegate void Test(string name);
```

- We are declaring delegate nothing but function prototype which does not return anything.
- And its takes single parameter.
- Using this delegate we can work with all the methods whose return type is 'void' and the parameter is 'string'.
- Like we've show() method whose return type is 'void' and parameter is 'string'.
- We'll write another method display() which does not return anything and parameter is string type.

```
class Program
{
    delegate void Test(string name);

    static void Main(string[] args)
    {

        //Show("Manzoor");

        Console.ReadLine();
    }

    static void Show(string name)
    {
        Console.WriteLine("Show Hello "+name);
    }

    static void Display(string name)
    {
        Console.WriteLine("Display Hello " + name);
    }
}
```

- Both methods are different but prototype is same.
- So both this methods can be invoked with the help of delegate 'Test'.
- Now we'll create an object of delegate.

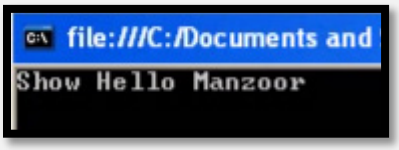
```
Test t = new Test(Show);
```

- Here we've given show method as parameter to be implement.
- Now it will be referring to show() method.
- And if we want invoke the method show() we can call it with the help of reference.

```
t.Invoke("Manzoor");
```

- And we've invoked that method using reference t and given parameter as "Manzoor".
- If we see that same thing we've done in earlier is statically but now we've done dynamically.

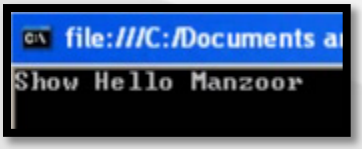
- This is runtime binding why because we are using 'new' keyword to allocate memory.
- Press F5.
- Now it's going to invoke show() method.



- Instead of invoking show method using 't.invoke' we can directly write as below.

```
//t.Invoke("Manzoor");
t("Manzoor");
```

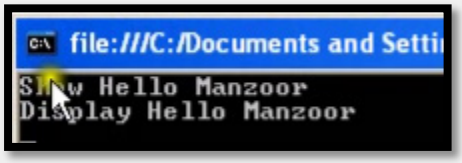
- Here 't' is acting like show() method.
- I.e. show of Manzoor.



- It displays same result.
- Now we will invoke Display() method.

```
t = new Test(Display);
t("Manzoor");
```

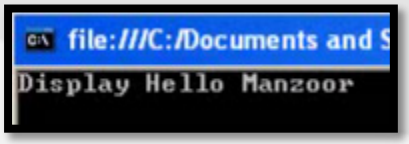
- Now our 't' pointer will be referring to Display() method.
- If we say t of Manzoor this time it will be calling Display() method.



- So first it was pointing to show method.
- And second it's pointing to Display method.
- So we are getting runtime polymorphism concept over here.
- That means we can make t an object of any method at runtime and we'll call t of message to execute that method at runtime.
- Code for this method is generate at runtime.
- This is the simple way of using delegate
- Let us see one more advantage of this.

```
Test t = new Test(Show);
//t.Invoke("Manzoor");
//t("Manzoor");

t = new Test(Display);
t("Manzoor");
```

- T earlier it was referring to show method.
- Now we made it refer to display method so it is going to work with display method.



- That's fine.
- Now what we can do here instead of changing the reference from show to display we can add it.

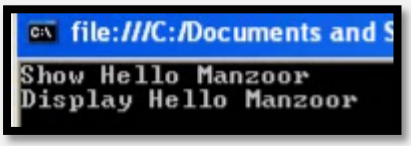
```
t += new Test(Display);
t("Manzoor");
```

- That means it was referring to show method now we've added one more reference of display to the same.

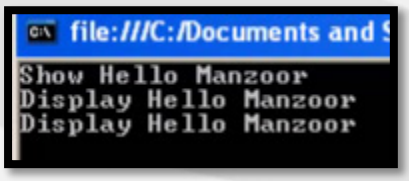
- Now ‘t’ will be referring to both methods show as well as display.
- Now if we say t of Manzoor what is that we’ll be expecting, we’ll be expecting it to execute display method.
- But we’ve used ‘+’ symbol that means now it is referring to show as well as display.

```
t("Manzoor");
```

- Once we call t of message it’s going to execute show method first then display method next.



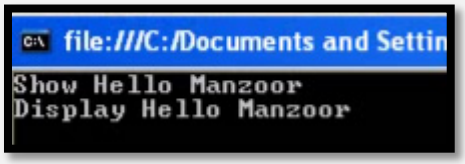
- This is the magic of delegate.
- If we add one more reference to the same display.
- Then it’s going to execute show method once and display method twice.



- One reference executing multiple things or multiple methods.
- With single line of code we are executing multiple methods.
- This concept is called as **multicasting** or **multicast delegates**.
- Now as we can add a reference in the way we can remove a reference.

```
Test t = new Test(Show);
//t.Invoke("Manzoor");
//t("Manzoor");

t += new Test(Display);
t += new Test(Display);
t -= new Test(Display);
```

- Here we've added display twice and removed once.
- Now it's going to display show method and display method once.



- This concept is called as **multicasting** or **multicast delegates**.
- If one reference is using for one method then it is called as **unicast delegates**.
- If one reference is using for many methods then it is called as **multicast delegated**.
- This is all about delegates.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegatesEg
{
    class Program
    {
        delegate void Test(string name);

        static void Main(string [] args)
        {
```

```
            //Show("Manzoor");

            Test t = new Test(Show);
            //t.Invoke("Manzoor");
            //t("Manzoor");

            t += new Test(Display);
            t += new Test(Display);
            t -= new Test(Display);

            t("Manzoor");



            Console.ReadLine();
        }

        static void Show(string name)
        {
            Console.WriteLine("Show Hello "+name);
        }

        static void Display(string name)
        {
            Console.WriteLine("Display Hello " + name);
        }
    }
}
```

CHAPTER 24 : DELEGATES, CALLBACK AND METHOD CONCEPT IN C#.

- In this chapter we're going to extend our earlier example.
- We saw two ways of calling a method.
- One is directly.
- And another way using delegates.
- We've two ways of calling the same method.
- And the difference is for direct we're generating the code at compile time.
- Whereas using delegates code is generated at runtime.
- Assume if we've a method already defined and if we want to call that method then we can go for direct way that means method definition is already done and we're calling that method.
- If we've a situation where method definition is not done that means we've not defined the method or we do not know the name of the method or we don't know the definition of the method but we know its return type and parameter.
- That means I knew the signature of the method but its implementation is not done and its name I don't know.
- That means till now we've seen the situation where method is defined prior to its invocation.
- That means we are defining the method first then we are invoking the method.
- If we've a situation where we need to invoke the method first later on we need to define wherever we need it.
- For example we've a button, button is nothing but we've a definition for class button.

- Now button has got an event say ‘we click the button’, so click is an event.
- That means onClick is a method we’ve defined inside the class button.
- Whenever we click the button that method gets invoke and that method intern will be calling a method which we haven’t defined yet.
- So whenever we’ve this kind of scenario where we need to invoke a method first later on in future anybody can define that method then we can go for ‘delegates’.
- This concept would be more clear when we go for events.

```
class Program
{
    delegate void Test(string name);

    static void Main(string[] args)
    {
        Show("Manzoor");

        Test t = new Test(Show);
        t("Manzoor");

        Console.ReadLine();
    }
    static void Show(string name)
    {
        Console.WriteLine("Show Hello " + name);
    }

    static void Display(string name)
    {
        Console.WriteLine("Display Hello " + name);
    }
}
```

- Now we’ll take delegate outside the class.
- And we’ll define a new class ‘Sample’.
- In this class we’ll create a reference for delegate.

```
class Sample
{
    Test t1;
}
```

- We've the variable or reference for that delegate as t1.
- And one method as 'CallBack' and the parameter we're going to pass is nothing but our delegate type.
- And we'll be invoking the method.

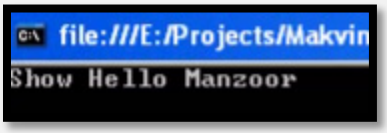
```
delegate void Test(string name);

class Sample
{
    Test t1;

    public void CallBack(Test t1)
    {
        this.t1 = t1;
        t1("Manzoor");
    }
}
```

- This is simple class in which we've one method 'CallBack' which takes parameter as type delegate 'Test'.
- And it is going to invoke the method.
- But we don't know which method it's going to invoke.
- Now let us create the object of class 'sample' in main method.
- And with that object we'll call the method 'CallBack' with parameter as object of delegate test i.e. t.

```
Test t = new Test(Show);
Sample s = new Sample();
s.CallBack(t);
```

- Once we pass t, t is referring to show now t1 will also referring to show.

- And we'll be invokes show method.
- Now execute it.



- Fine its invoking show method.
- Put a breakpoint.

```
Test t = new Test(Show);
Sample s = new Sample();
s.CallBack(t);
```

- Why we named as callBack, because from this class of main method we are calling callBack method of 'sample' class.

```
public void CallBack(Test t1)
{
    this.t1 = t1;
    t1("Manzoor");
}
```

- But this CallBack method intern calls the method of same class.

```
static void Show(string name)
{
    Console.WriteLine("Show Hello " + name);
}
```

- So we are calling back the method of same class.
- Now if we change from show to display then it will shows display method.

```
Test t = new Test(Display);
Sample s = new Sample();
s.CallBack(t);
```

- Execute it.

```
file:///E:/Projects/Makvin_Lapt
Display Hello Manzoor
```

- This works fine.
- We've pre-defined class and calling the method.

```
delegate void Test(string name);

class Sample
{
    Test t1;

    public void CallBack(Test t1)
    {
        this.t1 = t1;
        t1("Manzoor");
    }
}
```

- T1 is a method and its calling the method whose return type is 'void' and parameter is 'string'.
- If we observe we're calling the method whose return type is 'void' and parameter is 'string', its definition it has not defined yet.
- That means we're invoking the method prior to its definition this is the reverse way.
- After that we're defining that method and then calling it back.
- We've one more way of calling back.
- The way is instead of creating the reference to display method and then passing that in CallBack.

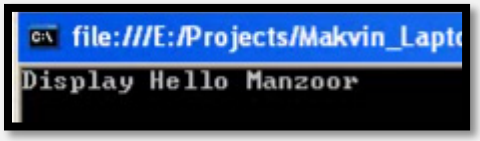
```
Test t = new Test(Display);
Sample s = new Sample();
s.CallBack(t);
```

- We can directly pass the method name which intern will pass its reference.

- Means we need not create separate delegate.

```
Sample s = new Sample();
s.CallBack(Display);
```

- Both are same.
- Press F5.



- We can also call one more method.

```
Sample s = new Sample();
s.CallBack(Display);
s.CallBack(Show);
```

- Now it'll be calling show method.

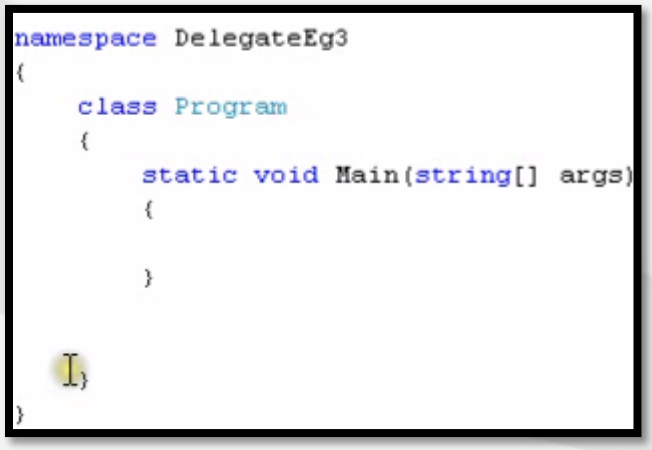
- In this concept we are invoking a method whose definition is not defined yet.
- So single line of code is invoking n number of methods whether it is show method or display method of same signature.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

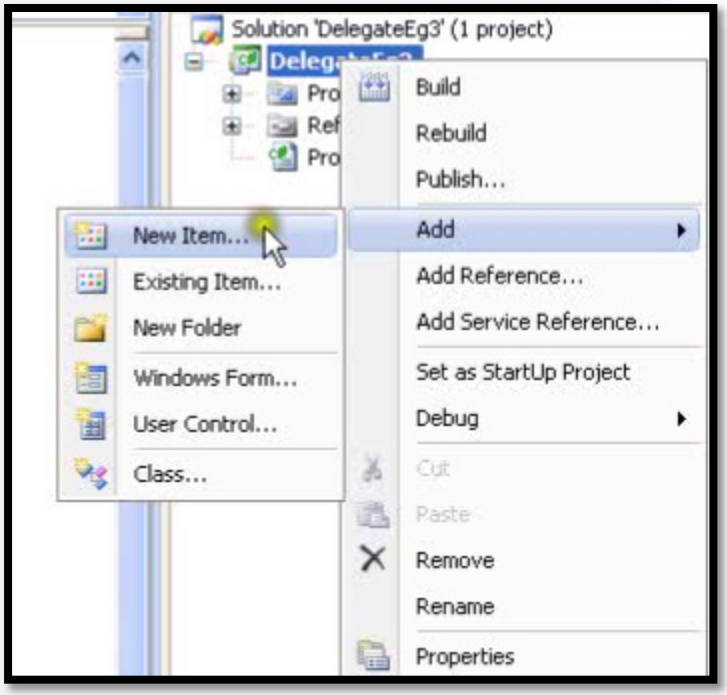
```
namespace DelegateEg2
{
    delegate void Test(string name);

    class Sample
    {
        Test t1;

        public void CallBack(Test t1)
        {
            this.t1 = t1;
            t1("Manzoor");
        }
    }

    class Program
    {


        static void Main(string [] args)
        {
            Sample s = new Sample();
            s.CallBack(Display);
            s.CallBack(Show);

            Console.ReadLine();
        }
        static void Show(string name)
```

```
        {
            Console.WriteLine("Show Hello " + name);
        }

        static void Display(string name)
        {
            Console.WriteLine("Display Hello " + name);
        }
    }
}
```

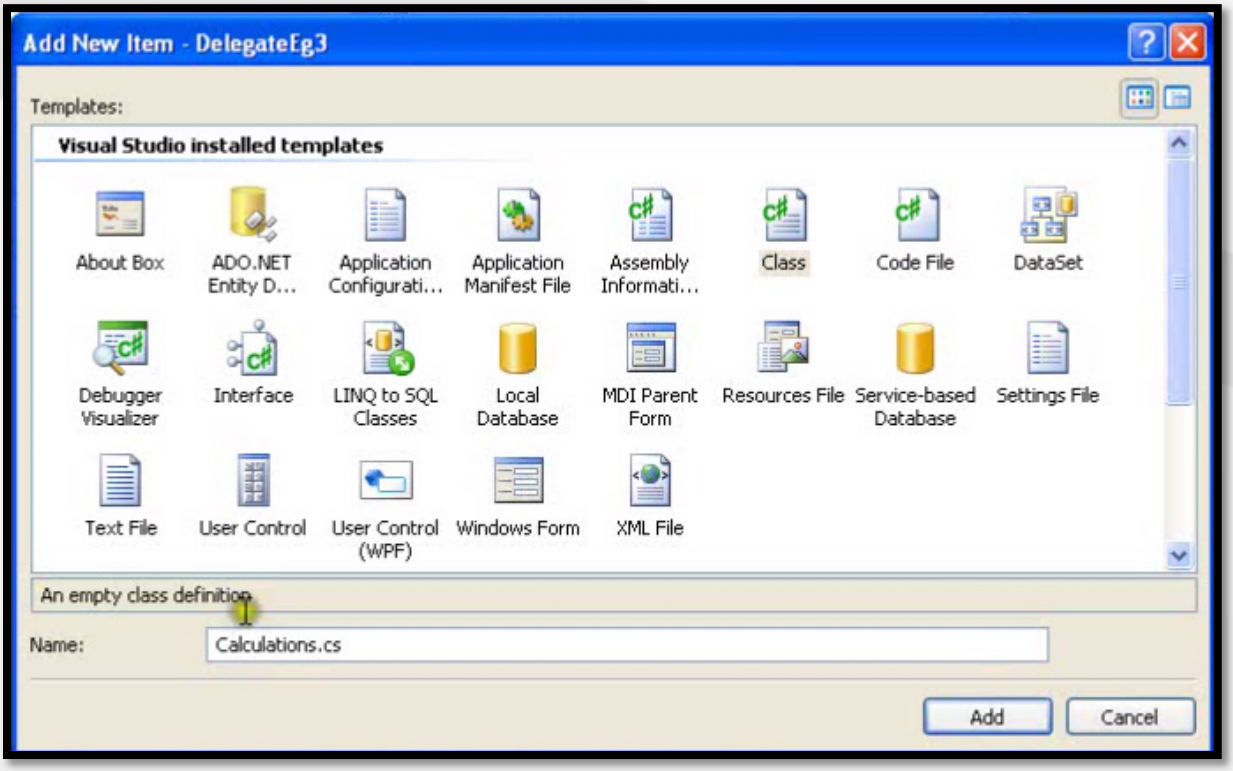
- We'll have one more example of delegates.



- This is our User Interface.
- Here we are going to work out on two layered architecture.
- This is our UI and our business logic layer will be in separate class.
- Now we'll add Business Logic Layer.



- Select on project→ Add→ New item.
- And select on class.



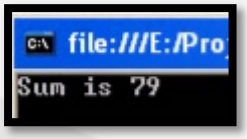
- Name it as ‘Calculations’.
- Press ok.
- In calculation class we’ve some code for arithmetic operations.

```
class Calculations
{
    public void Add(int a, int b)
    {
        Console.WriteLine("Sum is " + (a + b));
    }

    public void Sub(int a, int b)
    {
        Console.WriteLine("Diff is " + (a - b));
    }

    public void Product(int a, int b)
    {
        Console.WriteLine("Product is " + (a * b));
    }
}
```

- If we observe all the methods has same signature.
- Like same return type and same parameter list.
- So we can use delegates here.
- In the main program we’ll create a delegate ‘Test’ with two parameters.
- And we’ll create an object of calculations class.
- We’ll create a delegate object and in the constructor we’ll pass add method.
- If we pass two variables in the reference then it will add it and displays it.

```
namespace DelegateEg3
{
    delegate void Test (int a,int b);

    class Program
    {
        static void Main(string[] args)
        {
            Calculations c = new Calculations();
            Test t = new Test(c.Add);

            t(23, 56);
            Console.ReadLine();
        }


    }
}
```

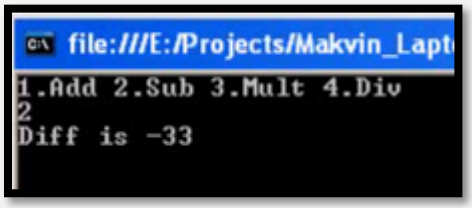
- Now it'll works for addition method.
- Press F5.



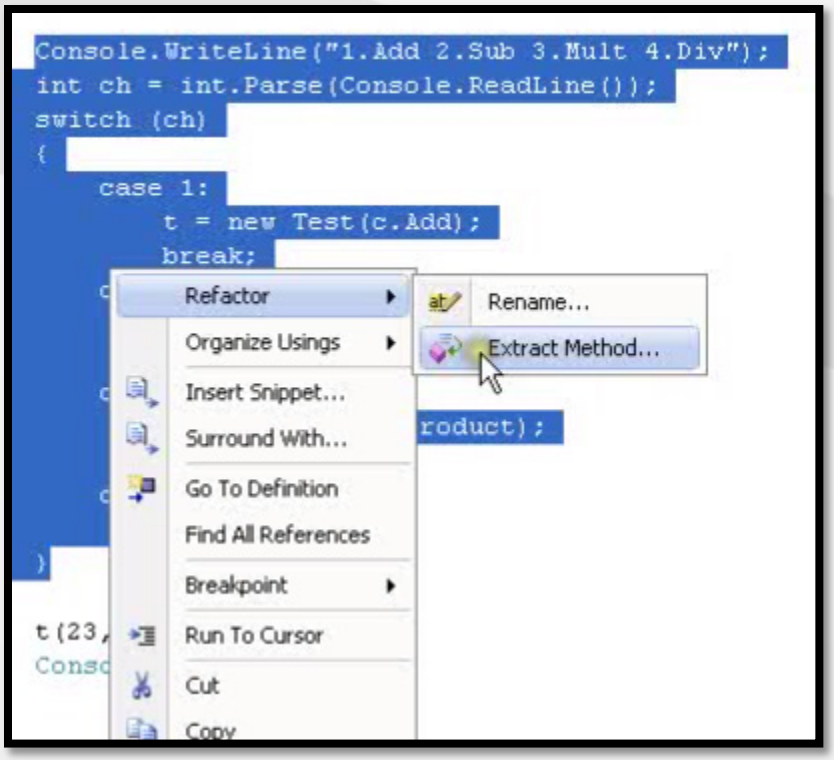
- This works fine.
- Now we'll give the options for end user to enter the choice.

```
Calculations c = new Calculations();
Test t = new Test(c.Add);

Console.WriteLine("1.Add 2.Sub 3.Mult 4.Div");
int ch = int.Parse(Console.ReadLine());
switch (ch)
{
    case 1:
        t = new Test(c.Add);
        break;
    case 2:
        t = new Test(c.Sub);
        break;
    case 3:
        t = new Test(c.Product);
        break;
    default:
        break;
}

t(23, 56);
Console.ReadLine();
```

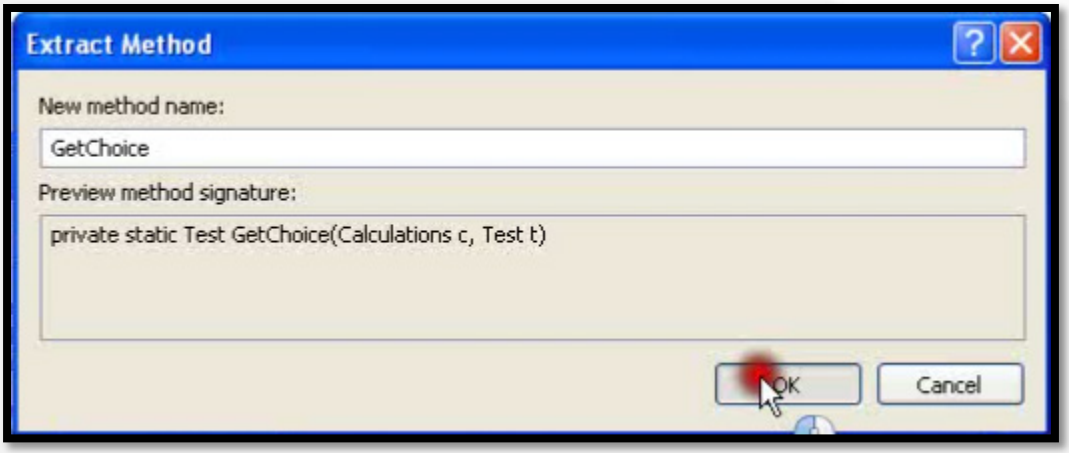
- Here we are calling the same method at runtime it is going to make 't' a reference of any method and it is going to execute it.
- That means we are calling a method for that name we haven't decided yet.
- That decision is made at runtime.
- Let us execute this.



- It's showing the options, we pressed 2.
- Then it's displaying the difference.
- In this program choices option with switch part take out to new method from main method using refactor.



- And name it as 'GetChoice'.



- Press ok.

```
static void Main(string[] args)
{
    Calculations c = new Calculations();
    Test t = new Test(c.Add);

    t = GetChoice(c, t);

    t(23, 56);
    Console.ReadLine();
}

private static Test GetChoice(Calculations c, Test t)
{
    Console.WriteLine("1.Add 2.Sub 3.Mult 4.Div");
    int ch = int.Parse(Console.ReadLine());
    switch (ch)
    {
```

- We got the method.
- This is going to get the choice of method then it is going to execute this.
- This what our UI is extracted method is in business logic layer.
- In future we can add new method, now we'll add a method as Division.

```
public void Product(int a, int b)
{
    Console.WriteLine("Product is " + (a * b));
}

public void Division(int a, int b)
{
    Console.WriteLine("Division is " + (a / b));
}
```

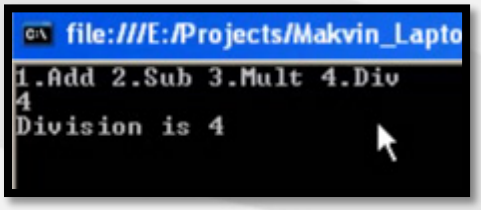
- In switch we need to add one more case for division.
- If we observe we're not at all touching the UI.

```
static void Main(string[] args)
{
    Calculations c = new Calculations();
    Test t = new Test(c.Add);

    t = GetChoice(c, t);

    t(23, 56);
    Console.ReadLine();
}
```

- UI is same, just we are working on business logic layer.

```
private static Test GetChoice(Calculations c, Test t)
{
    Console.WriteLine("1.Add 2.Sub 3.Mult 4.Div");
    int ch = int.Parse(Console.ReadLine());
    switch (ch)
    {
        case 1:
            t = new Test(c.Add);
            break;
        case 2:
            t = new Test(c.Sub);
            break;
        case 3:
            t = new Test(c.Product);
            break;
        case 4:
            t = new Test(c.Division);
            break;
        default:
            break;
    }
    return t;
}
```

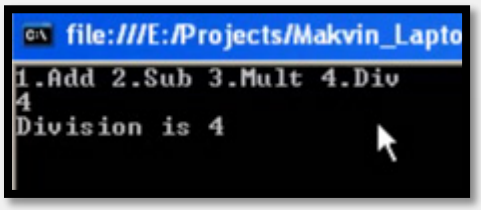
- Press F5.



- Here we've selected 4.
- So it displays the division result.
- This is all about uses and implementation of delegates.
- Delegates are nothing but pointers to the methods or references to the methods.
- So this is all about delegates.
- Complete code is given below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DelegateEg3
{
    delegate void Test (int a,int b);

    class Program
    {
        static void Main(string [] args)
        {
            Calculations c = new Calculations();
            Test t = new Test(c.Add);

            t = GetChoice(c, t);

            t(230, 56);
            Console.ReadLine();
        }

        private static Test GetChoice(Calculations c, Test t)
        {
            Console.WriteLine("1.Add 2.Sub 3.Mult 4.Div");
            int ch = int.Parse(Console.ReadLine());
            switch (ch)
            {
                case 1:
```

```
                t = new Test(c.Add);
                break;
            case 2:
                t = new Test(c.Sub);
                break;
            case 3:
                t = new Test(c.Product);
                break;
            case 4:
                t = new Test(c.Division);
                break;
            default:
                break;
        }
        return t;
    }


}
}
```



- Thank you..!